Homomorphic Encryption

Bill Buchanan, Baraq Ghaleb, Sam Grierson

Outline

December 10, 2024

Edinburgh Napier UNIVERSITY
School of Computing, Engineering & the Built Environment

Contents

Chapter 1

Introduction

This document is split into 12 main chapters:

1. Background.
2. State of the Art Methods.
3. Examine secure data sharing.
4. Cloud-based operations.
5. Secure multi-party computation.
6. Machine learning applications.
7. Modern implementations of HE.
8. Quantum resilience.
9. Hardware requirements.
10. Data fragmentation.
11. Side Channel Analysis.
12. Additional research area.

The delivery of milestones are:

1. Milestone 1 (Sep 2024): Delivery of chapters 1 and 2.
2. Milestone 2 (4 Oct 2024): Delivery of chapters 4, 5, 6 and 7.
3. Milestone 3 (6 Dec 2024): Delivery of the rest of the chapters.

1.1 Introduction

In 1978, Rivest, Adleman, and Dertouzos [1] published a paper that noted an interesting property of their seminal work on the Rivest-Shamir-Adleman (RSA) public key cryptosystem [2]. The paper called this property a privacy homomorphism, which essentially allowed two naïve RSA ciphertexts to be multiplied, multiplying the underlying plaintexts. At the end of their paper, Rivest, Adleman, and Dertouzos pose an open question, asking whether achieving privacy homomorphisms with large sets of operations that satisfy computational security definitions would be possible.

For around 30 years, various authors have proposed schemes that satisfy common cryptographic security definitions and feature privacy homomorphisms as an additional property. Some examples include the Elgamal cryptosystem with a multiplicative homomorphism [3], Paillier's Public Key Encryption (PKE) scheme with an additive homomorphism [4], and Goldwasser and Micali's encryption scheme with an "exclusive or" homomorphism [5]. However, during this period, it remained unclear whether achieving privacy homomorphisms with an arbitrary number of operations was possible. The first positive sign came more than 20 years later when Sander, Young, and Yung published their 1999 paper detailing a cryptosystem capable of evaluating logarithmic depth computational circuits [6]. Several schemes followed the example of Sander, Young, and Yung, including Boneh, Goh, and Nissim, who proposed a scheme based on their work with pairing groups that can evaluate an arbitrary number of additions and a single multiplication on ciphertexts [7].

After 30 years of attempts to answer Rivest, Adleman, and Dertouzs open question, in 2009 Gentry [8] proposed what can be considered the first plausible construction of what is now called a Fully Homomorphic Encryption (FHE) scheme. Gentry's construction was simple but brilliant; he first took a scheme restricted to evaluating low-degree polynomials and modified it using a new technique he developed. In his PhD thesis, Gentry chose a scheme based on ideal lattice assumptions [9, 10, 11] capable of evaluating addition and multiplication operations on ciphertexts until a build-up of noise would cause incorrect decryptions. Gentry then showed that homomorphically evaluating the decryption operation on noisy ciphertexts would reduce the noise, allowing further operations. This process, called bootstrapping, has become the *de facto* method for building FHE. The implementation of Gentry's scheme by himself and Halevi [12] was incredibly inefficient by modern standards but proved that FHE schemes were indeed possible and not just on paper. Later, van Dijk *et al.* [13] took the methods proposed by Gentry and showed they were not restricted to schemes based on ideal lattice

assumptions.

1.1.1 Four Generations of FHE

The approach taken by Gentry [8] and van Dijk *et al.* [13] is considered to be the first generation of FHE. The second generation of FHE began around 2011, with work focused on the practical realisation of proposed FHE schemes. The first of these schemes was that of Brakerski, Gentry, and Vaikuntanathan (BGV) [14], which iterated earlier work by Brakerski and Vaikuntanathan [15] on levelled FHE using modulus switching and relinearisation. The approach to FHE by the BGV scheme also differed from Gentry's original scheme in its security assumption, instead relying on the well-studied Learning With Errors (LWE) [16] and Ring LWE (RLWE) assumptions [17]. Later, the BGV scheme was modified by work by Fan and Vaikuntanathan [18] based on the scale-invariant FHE work of Brakerski [19] to enable practical levelled HE without the need for regular modulus switching in a scheme dubbed BFV. Further work by Gentry, Halevi, and Smart [20] illustrated that FHE can be achieved with polylog overhead using BGV and BGV alongside a technique they developed called ciphertext packing. While BGV and BFV became the more successful FHE schemes proposed around this time, there was parallel research into FHE schemes based on the NTRU [9] assumption. Unfortunately, the variant of NTRU used by these schemes was shown to be insecure [21, 22], so we avoid discussing these schemes in great detail.

The third generation of FHE began with the proposal of a scheme by Gentry, Sahai, and Waters (GSW) in 2013 [23], which avoided the expensive relinearisation step involved in second-generation FHE schemes. Furthermore, Brakerski and Vaikuntanathan found that the GSW scheme had an even slower rate of noise growth than previously proposed FHE schemes [24], which Alperin-Sheriff and Peikert used to propose an efficient bootstrapping method [25]. The original GSW scheme was then converted from a strictly LWE-based scheme into an RLWE-based scheme in two different proposals: Ducas and Micciancio's FHEW [26] and Chillotti *et al.*'s TFHE [27]. The FHEW scheme was the first FHE scheme to show that by refreshing ciphertexts after every operation, performing bootstrapping could be reduced to a negligible amount of overhead. The TFHE scheme further improved upon FHEW's success by implementing a ring variant of FHEW's bootstrapping method.

The fourth, and thus far final, generation consists of a single FHE scheme proposed by Cheon, Kim, Kim, and Song (CKKS) in 2016 [28]. The key advance of the CKKS scheme is its support of rounding operations, enabling

the support of a special kind of fixed-point arithmetic. Additionally, the proposed rescaling operation helps reduce the noise buildup during multiplications of ciphertexts, further reducing the bootstrapping required by a high-depth computation. Due to the rescaling of ciphertexts, CKKS is one of the most efficient methods for evaluating approximate arithmetic circuits. The security of the CKKS scheme is based on RLWE, which has reductions similar to the BGV and BFV. However, in 2020, Li and Micciancio published a paper that showed that passive attacks were possible against CKKS, with the suggestion that the common INDistinguishability against Chosen Plaintext Attacks (IND-CPA) may not be a sufficient security definition for CKKS [29]. Recently, work has been published providing strategies to mitigate the security threat posed by Li and Micciancio's attack, most notably in work by Cheon, Hong, and Kim [30].

1.1.2 Organisation

The rest of this chapter is structured as follows: Section 1.2 provides an overview of the formal properties an encryption scheme must satisfy to be considered a $\mathcal{F}$-HE scheme, along with various classifications of HE schemes.

1.2 Homomorphic Encryption

In this section, HE and its various properties will be formally defined, along with the security requirements that a HE scheme would be required to fulfil. As discussed previously, HE is a form of encryption with additional properties that allow evaluation of a function over encrypted data without access to a secret key. We first begin by defining a plaintext space $\mathcal{M} \stackrel{\text{def}}{=} \{0,1\}^*$ and a family of functions $\mathcal{F} \stackrel{\text{def}}{=} \{f_i\}_{i=1}^n$ where the functions in this space are defined as $f_i : \mathcal{M} \to \mathcal{M}$. A function $f \in \mathcal{F}$ can be expressed in terms of a Boolean circuit, which is a directed acyclic graph where vertices are unary or binary Boolean gates, and edges are input in $\mathbb{Z}_2$. We can now define a $\mathcal{F}$-HE scheme, which follows the work of Brakerski and Vaikuntanathan [15] where they define a circuit evaluation scheme over encrypted inputs.

Definition 1 ($\mathcal{F}$-Homomorphic Encryption) *Let $\mathcal{M}$ be a plaintext space and $\mathcal{F}$ be a family of functions over the plaintext space. A* $\mathcal{F}$-HE *scheme for $\mathcal{F}$ is a tuple of* PPT *algorithms* $\mathsf{HE} \stackrel{\text{def}}{=} (\mathsf{gen}, \mathsf{enc}, \mathsf{eval}, \mathsf{dec})$ *such that*

$$(\mathsf{pk}, \mathsf{sk}, \mathsf{ek}) \xleftarrow{\$} \mathsf{gen}(1^\lambda, \alpha) \qquad (1.1)$$

$$c \xleftarrow{\$} \mathsf{enc}_{\mathsf{pk}}(m) \qquad (1.2)$$

$$c \xleftarrow{\$} \mathsf{eval}_{\mathsf{ek}}(f, [c_1, \ldots, c_n]) \qquad (1.3)$$

$$m = \mathsf{dec}_{\mathsf{sk}}(c) \qquad (1.4)$$

where $\mathcal{X}$ is the space of freshly encrypted ciphertexts, $\mathcal{Y}$ is the space of evaluation outputs, $\lambda \in \mathbb{N}$ is a security parameter, and $\alpha \in \{0,1\}^*$ is an auxiliary input.

Note that in the above definition the algorithms eval and dec can take inputs from both the fresh ciphertext space $\mathcal{X}$ and the evaluation ciphertext space $\mathcal{Y}$. As such, we also define $\mathcal{Z} \stackrel{\text{def}}{=} \mathcal{X} \cup \mathcal{Y}$ as the union of both spaces for convenience. Additionally, ek is frequently part of pk, but defining the scheme as above in the case where $\mathsf{ek} = \mathsf{pk}$ is not expressly forbidden. However, it should be noted that the separation of ek and pk is becoming a standard definition for HE [15].

1.2.1 Properties

Using the definition of a $\mathcal{F}$-HE scheme, we can define some properties a candidate needs to meet to be considered a valid HE scheme. The first property is the correctness of the scheme itself, *i.e.* a scheme must be able to decrypt a ciphertext to the correct plaintext without error and with high probability the evaluation of a function $f \in \mathcal{F}$ must yield a correct result.

Definition 2 (Correctness) *Let pk, sk, and ek be the output of $\mathsf{gen}(1^\lambda, \alpha)$ for a suitable $\lambda \in \mathbb{N}$ and $\alpha \in \{0,1\}^*$. An $\mathcal{F}$-HE scheme satisfies correct decryptions if $\forall m \in \mathcal{M}$*

$$\Pr[\mathsf{dec}_{\mathsf{sk}}(\mathsf{enc}_{\mathsf{pk}}(m)) = m] = 1$$

and correct evaluation if $\forall f \in \mathcal{F}$ and $\forall c_i \in \mathcal{X}$ such that $m_i = \mathsf{dec}_{\mathsf{sk}}(c_i)$ where $i \in [n]$, there exists some negligible function $\varepsilon : \mathbb{N} \to [0,1]$ such that

$$\Pr[\mathsf{dec}_{\mathsf{sk}}(\mathsf{eval}_{\mathsf{ek}}(f, [c_1, \ldots, c_n])) = f(m_1, \ldots, m_n)] = 1 - \varepsilon(\lambda).$$

For a HE scheme to be useful, a property we require is that ciphertext sizes do not grow significantly with homomorphic operations and that the output length of an evaluation depends on the security parameter. This property effectively rules out the trivial HE scheme where an evaluation is the identity function that outputs $(f, [c_1, \ldots, c_n])$ and the decryption algorithm decrypts $c_1, \ldots, c_n$ and applies f to them. The definition of compactness from Gentries thesis [8] was slightly different from the one given below, which is based on the later work of van Djik *et al.* [13] and more straightforward.

Definition 3 (Compactness [13]) *Let* ek *be the output of* $\mathsf{gen}(1^\lambda, \alpha)$ *for a suitable* $\lambda \in \mathbb{N}$ *and* $\alpha \in \{0,1\}^*$. *A* $\mathcal{F}$-HE *scheme is compact if* $\forall f \in \mathcal{F}$ *and* $\forall c_i \in \mathcal{Z}$ *where* $i \in [n]$ *then* $|\mathsf{eval}_{\mathsf{ek}}(f, [c_1, \ldots, c_n])| \leq \text{poly}(\lambda)$ *independent of the size of the circuit evaluating* f.

The final property is circuit privacy, which can be easily confused with the notion of circuit obfuscation since both appear to ensure the privacy of the circuit. However, circuit obfuscation deals with concealing the circuit and is useful when a circuit computes an algorithm that itself may be valuable and must be hidden. On the other hand, circuit privacy is a characterisation of the distributions of outputs of the enc and eval algorithms. The property of circuit privacy has sometimes been known as the strongly homomorphic property in the literature [31], which has proved to be a point of contention regarding the correct way to define this property. In this document, we use the original definition given by Gentry [8].

Definition 4 (Circuit Privacy [8]) *Let* pk, sk, *and* ek *be the output of* $\mathsf{gen}(1^\lambda, \alpha)$ *for a suitable* $\lambda \in \mathbb{N}$ *and* $\alpha \in \{0,1\}^*$. *A* $\mathcal{F}$-HE *scheme is computationally circuit private if* $\forall f \in \mathcal{F}$ *and all* $\forall c_i \in \mathcal{Z}$ *such that* $m_i = \mathsf{dec}_{\mathsf{sk}}(x_i)$ *where* $i \in [n]$, *then*

$$\{\mathsf{eval}_{\mathsf{ek}}(f, [c_1, \ldots, c_n])\} \overset{c}{\approx} \{\mathsf{enc}_{\mathsf{pk}}(f(m_1, \ldots, m_n))\}$$

where both distributions are taken over the randomness of their respective algorithms.

The above definition may not be immediately evident as to why it implies that a circuit must be private. Intuitively, the definition states that the output of the evaluation for a function on ciphertexts is computationally hard to distinguish from the encrypted output of that function on plaintexts. In other words, since the encrypted output of the function evaluated on plaintexts is just another ciphertext, it is computationally difficult for an adversary to determine how it was generated.

1.2.2 Classifications

There are some additional properties that a $\mathcal{F}$-HE scheme can possess that allow them to be classified into different types of schemes depending on what type of functions they can evaluate. There are three broad categories that a HE scheme can fall into and we start by defining each formally based on their required properties.

Definition 5 (Somewhat Homomorphic Encryption) *A $\mathcal{F}$-HE scheme* HE *where $\mathcal{F} \neq \emptyset$ and* HE *satisfies the correctness property is called a Somewhat Homomorphic Encryption (SHE) scheme.*

Note that the set $\mathcal{F}$ need only consist of "some" functions, with no requirement on the type of functions the scheme can evaluate. Furthermore, compactness is not required so that ciphertexts may increase significantly with each homomorphic operation.

The next type of scheme is a levelled HE scheme. There can be some points of confusion between a SHE scheme and a levelled HE scheme. In the levelled HE case, other than the depth d of the circuit that evaluates $f \in \mathcal{F}$, there are no restrictions on the set $\mathcal{F}$. With SHE, d may be varied by additional parameter choices, most notably the size of ciphertexts. Conversely, levelled HE schemes define a specific value of d based on the input parameter independent of the ciphertext size. The first to propose the construction of levelled HE schemes was Brakerski and Vaikuntanathan [15], and the following definition is based on that given in their original work.

Definition 6 (Levelled Homomorphic Encryption [15]) *A $\mathcal{F}$-*HE *scheme is called a levelled homomorphic encryption scheme if it takes an auxiliary input $\alpha = d \in \mathbb{N}$ to* gen *specifying the maximum depth of a circuit representation of $f \in \mathcal{F}$ that can be evaluated. Additional requirements are made for correctness, such that the length of the evaluation output does not depend on d.*

The parameter α was introduced into the gen functions in the definition of HE specifically for allowing the definition of the maximum depth of a circuit that evaluates $f \in \mathcal{F}$. In the case of levelled HE, the assumption that $\alpha = \text{poly}(\lambda)$ can be justified since $\alpha = d$ is constant. However, the goal of our definitions here is to be as general as possible. Therefore, there may be cases in which α may be greater than $\text{poly}(\lambda)$.

The final definition is FHE, the ultimate type of HE scheme and a holy grail of cryptography. A scheme that satisfies the definition of a FHE scheme can evaluate any $f \in \mathcal{F}$ using a circuit of arbitrary depth, and where $\mathcal{F}$ is the set of any computable function.

Definition 7 (Fully Homomorphic Encryption) *A* FHE *scheme is a $\mathcal{F}$-*HE *scheme that is compact, correct, and where $\mathcal{F}$ is the set of all computable functions.*

There is one additional property that can be used to classify a HE scheme: the notion of computation stages. In some cases, it may be necessary to

compute a function $f \in \mathcal{F}$ homomorphically over two or more stages. If this is the case, we would need to take the output of eval and any additional ciphertexts output by enc necessary to evaluate f and use them as inputs for eval. In the earlier definition of evaluation correctness, the only guarantees made are that the algorithm eval is correct for inputs in $\mathcal{X}$. We now want to consider conditions under which an evaluation algorithm works given evaluation outputs.

Homomorphically evaluating a function $f \in \mathcal{F}$ over multiple stages is called i-hop HE, coined by Gentry, Halevi, and Vaikuntanathan [32] and Rothblum [31], where either $i \in \mathbb{N}$ or the scheme is categorised as multi-hop. Before formally defining the notion of i-hop correctness, we will give a formal definition for a staged computation. A function $F_{i,n}$ evaluated in i stages of input length n can be defined by a set of functions $F_{i,n} \stackrel{\text{def}}{=} \{f_{k\ell}\}_{1\leq k\leq i, 1\leq \ell\leq n}$ where $f_{k\ell}$ has kn inputs. Given a set of inputs $\mathbf{m}_1 \stackrel{\text{def}}{=} [m_{1,1}, \ldots, m_{1,n}] \in \mathcal{M}^n$ the computation can be denoted by

$$m_{k\ell} = f_{k\ell}(m_{1,1}, \ldots, m_{1,n}, \ldots, m_{k,1}, \ldots, m_{k,n})$$

for $1 \leq k \leq i$ and $1 \leq \ell \leq n$. The output of a staged evaluation would be denoted by $\mathbf{m}_i \stackrel{\text{def}}{=} [m_{i,1}, \ldots, m_{i,n}] \in \mathcal{M}^n$. We use the more compact notation $\mathbf{m}_i = F_{i,n}(\mathbf{m}_1)$ for a staged computation. Now, given pk, ek, and sk output by the algorithm gen and an input ciphertext $\mathbf{c}_1 \stackrel{\text{def}}{=} [c_{1,1}, \ldots, c_{1,n}] \in \mathcal{X}^n$ we denote a staged homomorphic evaluation as $\mathbf{c}_i = \mathsf{eval}_{\mathsf{ek}}(F_{i,n}, \mathbf{c}_1)$ where $\mathbf{c}_i \stackrel{\text{def}}{=} [c_{i,1}, \ldots, c_{i,n}] \in \mathcal{X}^n$. Additionally, we introduce the notation $\mathbf{m} = \mathsf{dec}_{\mathsf{sk}}(\mathbf{c})$ where $\mathbf{m} \in \mathcal{M}^n$ and $\mathbf{c} \in \mathcal{X}^n$, which is defined by $\mathbf{m}[i] = \mathsf{dec}_{\mathsf{sk}}(\mathbf{c}[i])$ for $1 \leq i \leq n$. With these new notations, we can now define the notation of i-hop correctness derived from work by Gentry, Halevi, and Vaikuntanathan [32] and Rothblum [31].

Definition 8 (i-Hop Correctness [32, 31]) *Let* pk, sk, *and* ek *be the output of* $\mathsf{gen}(1^\lambda, \alpha)$ *for a suitable* $\lambda \in \mathbb{N}$ *and* $\alpha \in \{0,1\}^*$. *Given any staged computation* $F_{i,n}$ *where* $n = \text{poly}(\lambda)$ *a* $\mathcal{F}$-HE *scheme is* i-*hop correct if* $\forall \mathbf{c}_1 \in \mathcal{X}^n$ *such that* $\mathbf{m}_1 = \mathsf{dec}_{\mathsf{sk}}(\mathbf{c}_1) \in \mathcal{M}^n$ *there exists some negligible function* $\varepsilon : \mathbb{N} \to [0,1]$ *such that*

$$\Pr[\mathsf{dec}_{\mathsf{sk}}(\mathsf{eval}_{\mathsf{ek}}(F_{i,n}, \mathbf{c}_1)) = F_{i,n}(\mathbf{m}_1)] = 1 - \varepsilon(\lambda).$$

Using the definition of i-hop correctness, HE schemes can be further divided into new classifications based on their capabilities to perform a certain number of stages of staged computation. The following definitions were derived from work by Gentry, Halevi, and Vaikuntanathan [32] and Rothblum [31].

Definition 9 (i-Hop [32, 31]) *A $\mathcal{F}$-*HE *scheme* HE *is i-hop for $i \in \mathbb{N}$ if j-hop correctness holds for $\forall j \in \mathbb{N}$ where $1 \leq j \leq i$.*

Definition 10 (Multi-Hop [32, 31]) *A $\mathcal{F}$-*HE *scheme* HE *is multi-hop if j-hop correctness holds for $\forall j \in \mathbb{N}$ where $1 \leq j \leq \text{poly}(\lambda)$.*

With these definitions, the relationship between FHE and i-hop may cause some confusion. It may seem natural that if it is possible to evaluate an arbitrary function, it is possible to evaluate arbitrarily many functions sequentially, as the composition of functions would indicate. Unfortunately, this is not the case. There are no guarantees that the output of an algorithm implementing the eval function look anything like a freshly encrypted ciphertext and that they are valid inputs for eval.

1.2.3 Security

Security of a HE scheme has been typically defined by the IND-CPA security notion first proposed by Goldwasser and Micali [5] in their seminal work. The intuitive notion of IND-CPA is that an encryption scheme is secure if no PPT adversary has an advantage negligibly greater than $\frac{1}{2}$ when attempting to distinguish whether a given ciphertext is an encryption of two equally likely and distinct messages. The key to achieving IND-CPA security is the randomisation of ciphertexts, resulting in two different encryptions of the same plaintext being indistinguishable. The reason IND-CPA suffices for HE schemes is that for the majority of applications of HE, encrypted data that remains confidential is only accessible to the owner of the decryption key, which is what IND-CPA security guarantees.

Definition 11 (IND-CPA Security [5]) *Let* HE *be a HE scheme and* $\text{Expr}^{\text{cpa}}[\mathcal{A}, \mathsf{HE}]$ *be a probabilistic experiment defined as*

$$
\begin{array}{l}
\underline{\text{Expr}^{\text{cpa}}[\mathcal{A}, \mathcal{E}](\lambda):} \\
(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{gen}(1^\lambda) \\
(m_0, m_1) \xleftarrow{\$} \mathcal{A}(\mathsf{pk}) \\
b \xleftarrow{\$} \{0, 1\} \\
c \xleftarrow{\$} \mathsf{enc}_{\mathsf{pk}}(m_b) \\
b' \overset{\text{def}}{=} \mathcal{A}(\mathsf{pk}, c) \\
\texttt{return } 1 \texttt{ if } b = b' \\
\texttt{else } 0
\end{array}
$$

for a PPT *adversary $\mathcal{A}$ and a* PKE *scheme $\mathcal{E}$. The scheme* HE *is* IND-CPA *secure if for all* PPT *adversaries $\mathcal{A}$ and a suitable $\lambda \in \mathbb{N}$ there exists a*

negligible function $\varepsilon : \mathbb{N} \to [0, 1]$ *such that*

$$\mathrm{Adv}^{\mathrm{cpa}}_{\mathcal{A},\mathsf{HE}}(\lambda) \stackrel{\mathrm{def}}{=} \Pr\left[\mathrm{Expr}^{\mathrm{cpa}}[\mathcal{A}, \mathsf{HE}](\lambda) = 1\right] \leq \frac{1}{2} + \varepsilon(\lambda).$$

It should be noted that while IND-CPA may be a sufficient security definition for many cases, and is the *de facto* standard for HE schemes, it does not require ciphertext integrity. There are stronger definitions of security are possible for HE schemes, the two most common for encryption schemes are INDistinguishability under non-adaptive Chosen Ciphertext Attacks (IND-CCA1) and INDistinguishability under adaptive Chosen Ciphertext Attacks (IND-CCA2) which prevent attacks derived from an adversaries ability to modify a ciphertext [33, 34]. Unfortunately, IND-CCA2 security is unachievable for is not achievable for HE schemes since a homomorphism is a direct contradiction to the whole definition of security. However, there have been attempts to build HE schemes that achieve IND-CCA1 security, with varying levels of success [35, 36]. In this document, we only consider IND-CPA security since it is the security definition used in most proposed HE schemes [14, 18, 23, 28].

Chapter 2

Background

2.1 Introduction

This chapter outlines some of the fundamental principles involved with homomorphic encryption, including the usage of polynomials and learning with errors (LWE). With homomorphic encryption, we can take plaintext of A and B and produce:

$$Enc_k(A \circ B) = Enc_k(A) \circ Enc_k(B) \tag{2.1}$$

and where $\circ$ can be any arithmetic operator such as add, subtract, multiply and divide. With symmetric key encryption, we use the same key to decrypt as we do to encrypt.

2.2 Partial homomorphic

With partial homomorphic encryption (PHE), we can implement some form of arithmetic operation in a homomorphic way. These methods include Paillier [37], ElGamal and a modified RSA version.

2.2.1 Paillier

The Paillier cryptosystem is a partial homomorphic encryption (PHE) method and can perform addition, subtraction and scalar multiply. Thus we get:

$$Enc_k(A + B) = Enc_k(A) + Enc_k(B) \tag{2.2}$$
$$Enc_k(A - B) = Enc_k(A) - Enc_k(B) \tag{2.3}$$
$$Enc_k(A.B) = A.Enc_k(B) \tag{2.4}$$

If we take two values: m_1 and m_2, we get two encrypted values of $Enc(m_1)$ and $Enc(m_2)$. We can then multiply the two cipher values to get $Enc(m_1 + m_2)$. We can then decrypt to get $m_1 + m_2$. Along with this, we can also subtract to $Enc(m_1 - m_2)$. This is achieved by taking the inverse modulus of $Enc(m_2)$ and multiplying it with $Enc(m_1)$. Finally, we can perform a scalar multiply to get $Enc(m_1 \cdot m_2)$ and which is generated from $Enc(m1)^{m2}$.

First we select two large prime numbers (p and q) and compute:

$$N = pq \tag{2.5}$$
$$PHI = (p-1)(q-1) \tag{2.6}$$
$$\lambda = \operatorname{lcm}(p-1, q-1) \tag{2.7}$$

and where lcm is the least common multiple. We then select a random integer g for:

$$g \in \mathbb{Z}^*_{nN^2} \tag{2.8}$$

We must make sure that n divides the order of g by checking the following:

$$\mu = (L(g^\lambda \pmod{n})^2))^{-1} \pmod{N} \tag{2.9}$$

and where L is defined as $L(x) = \frac{x-1}{N}$. The public key is (N, g) and the private key is (λ, μ).

To encrypt a message (M), we select a random r value and compute the ciphertext of:

$$c = g^m \cdot r^N \pmod{N^2} \tag{2.10}$$

and then to decrypt:

$$m = L(c^\lambda \pmod{N})^2) \cdot \mu \pmod{N} \tag{2.11}$$

For adding and scalar multiplying, we can take two ciphers (C_1 and C_2), and get:

$$C_1 = g^{m_1} \cdot r_1^N \pmod{N^2} \tag{2.12}$$
$$C_2 = g^{m_2} \cdot r_2^N \pmod{N^2} \tag{2.13}$$

If we now multiply them, we get:

$$C_1 \cdot C_2 = g^{m_1} \cdot r_1^N \cdot g^{m_2} \cdot r_2^N \pmod{N^2} \tag{2.14}$$
$$C_1 \cdot C_2 = g^{m_1+m_2} \cdot r_1^N \cdot r_2^N \pmod{N^2} \tag{2.15}$$

And when adding two values requires a multiplication of the ciphers. Examples of using Paillier are defined at [38].

2.2.2 ElGamal - Multiply and Divide

With ElGamal public key encryption, we have a public key of (Y, g, p) and a private key of (x). The cipher has two elements (a and b). With this, Bob selects a private key of x and then calculates Y ($g^x \pmod p$) for his public key. We can use it for partial homomorphic encryption, and where we can multiply the ciphered values. With this, we encrypt two integers and then multiply the ciphered values, and then decrypt the result [39].

Initially, Bob creates his public key by selecting a g value and a prime number (p) and then selecting a private key (x). He then computes Y which is:

$$Y = g^x \pmod p \tag{2.16}$$

His public key is (Y, g, p) and he will send this to Alice. Alice then creates a message (M) and selects a random value k). She then computes a and b:

$$a = g^k \pmod p \tag{2.17}$$

$$b = y^k M \pmod p \tag{2.18}$$

Bob then receives these and decrypts with:

$$M = \frac{b}{a^x} \pmod p \tag{2.19}$$

This works because:

$$\frac{b}{a^x} \pmod p = \frac{y^k M}{(g^k)^x} \pmod p \tag{2.20}$$

$$= \frac{(g^x)^k M}{(g^k)^x} \pmod p \tag{2.21}$$

$$= \frac{g^{xk} M}{g^{xk}} \pmod p \tag{2.22}$$

$$= M \tag{2.23}$$

2.2.3 RSA

RSA is a partially homomorphonic encryption method, and where we can add and multiply encrypted values. So with RSA we cipher with [40]:

$$Cipher = M^e \pmod N \tag{2.24}$$

and where e is the encryption key, and N is the multiplication of two prime numbers (p and q). The difficulty in this is factorizing N to get two prime numbers. To decrypt, we find the decryption key (d) and then perform:

$$M = Cipher^d \pmod{N} \tag{2.25}$$

So if we take two values (a and b), then we can cipher them as:

$$Cipher_1 = a^e \pmod{N} \tag{2.26}$$
$$Cipher_2 = b^e \pmod{N} \tag{2.27}$$

If we multiply the ciphers we get:

$$Cipher_3 = Cipher_1 \times Cipher_2 \tag{2.28}$$
$$= a^e \pmod{N} \times b^e \pmod{N} \tag{2.29}$$
$$= a^e \times b^e \pmod{N} \tag{2.30}$$

This is then:

$$Cipher_3 = (ab)^e \pmod{N} \tag{2.31}$$

When we can then decrypt and determine the value of a times b.

$$ab = (Cipher_3)^d \pmod{N} \tag{2.32}$$

2.3 Polynomials

With lattice methods, we have multi-dimensional spaces, and where to plot our points within there. Most of the time, we think in one dimension (x), or two dimensions (x, y), or three dimensions (x,y, and z), but in a lattice, we may have thousands of dimensions. Each point we have then fitted into a lattice of these points. To make a hard problem to solve, we add a small amount of error to the point. It is then extremely difficult to find the nearest point to the point with the error.

With lattices, we use polynomials to represent our multi-dimensional spaces. In general, a polynomial can be represented by a number of coefficients (a_n ... a_0) and polynomial powers:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ...a_1 x + a_0 \tag{2.33}$$

Within a lattice, we can have a point at (9, 5, 16), and then represent it with a quadratic equation of:

$$16x^2 + 5x + 9 \tag{2.34}$$

This could be represented by a polynomial of [16, 9, 5]. If we have another polynomial of $2x^2 + x + 7$, and multiply them together, we get:

$$f = (16x^2 + 5x + 9)(2x^2 + x + 7) = 32x^4 + 26x^3 + 135x^2 + 44x + 63 \quad (2.35)$$

In Python, we can determine this with:

```
from numpy.polynomial
import Polynomial
import numpy as np
f=Polynomial([16,9,5])g=Polynomial([2,1,7])
r = f*g
print(r.coef)
print(np.poly1d(r.coef))
```

And a sample run gives:

```
[ 32.  26. 135.  44.  63.]
    4       3        2
32 x + 26 x + 135 x + 44 x + 63
```

2.3.1 Mod polynomials

With polynomials, for an order of five, we have the form:

$$ax^5 + bx^4 + cx^3 + dx^2 + ex + f \quad (2.36)$$

This can then represent this as an integer array of:

$$[a, b, c, d, e, f] \quad (2.37)$$

In Python, the polynomial of $5x^4 + 4x^3 + x^2 + 11x + 10$ is implemented and printed as:

```
import numpy as np
A=[5,4,1,11,10]
print ("A:")
print(np.poly1d(A))
```

This gives:

```
A:
   4      3      2
5 x + 4 x + 1 x + 11 x + 10
```

Let's take an example of:

$$A = 10x^2 + 6x + 2B = 12x^2 + 3x + 2 \quad (2.38)$$

If we add these, we get:

$$A + B = 10x^2 + 6x + 2 + 12x^2 + 3x + 2 = 22x^2 + 9x + 4 \tag{2.39}$$

Within cryptography, we normally use finite fields, where the values are constrained by 0 and $p-1$, and where p is a prime number. For this, we can perform a $\pmod p$operation on this, such as with $\pmod{13}$:

$$A + B = 22x^2 + 9x + 4 = 9x^2 + 9x + 4 \pmod{13} \tag{2.40}$$

Now let's look at multiplication. For this, we have:

$$\begin{aligned} A \times B &= (10x^2 + 6x + 2) \times (12x^2 + 3x + 2) \\ &= 120x^4 + 30x^3 + 20x^2 + 72x^3 + 180x^2 + 12x + 24x^2 + 6x + 4 \\ &= 120x^4 + 102x^3 + 62x^2 + 18x + 4 \end{aligned}$$

And now, we can apply a $\pmod{13}$ operation to each of the coefficients:

$$A \times B = 120x^4 + 102x^3 + 62x^2 + 18x + 4 = 3x^4 + 11x^3 + 10x^2 + 5x + 4 \pmod{13}$$

Next, we will perform a polynomial subtraction:

$$A - B = (10x^2 + 6x + 2) - (12x^2 + 3x + 2) = -2x^2 + 3x \tag{2.41}$$

And now we take $\pmod{13}$:

$$A - B = -2x^2 + 3x = 11x^2 + 3x \pmod{13} \tag{2.42}$$

Next, we will perform a polynomial division:

$$A \div B = (10x^2 + 6x + 2)(12x^2 + 3x + 2) = 3x \tag{2.43}$$

And now $\pmod{13}$:

$$A \div B \pmod{13} = 3x \tag{2.44}$$

Code to implement this is [41]:

```
import numpy as np
import sys

q=13

A=[5,4,1,11,10]
```

```
B=[3,5,2,4]

if (len(sys.argv)>1):
    A=list(eval(sys.argv[1]))
if (len(sys.argv)>2):
    B=list(eval(sys.argv[2]))


print ("A:")
print(np.poly1d(A))
print ("B:")
print(np.poly1d(B))



print ("\n\n====== A+B =====")
res=np.polyadd(A,B)
print(np.poly1d(res))

res=np.polyadd(A,B)%q

print ("\nA+B (mod",q,"):")
print(np.poly1d(res))


print ("\n\n====== A*B =====")
res=np.polymul(A,B)
print(np.poly1d(res))

res=np.polymul(A,B)%q

print ("\n\nA*B (mod",q,"):")
print(np.poly1d(res))



print ("\n\n====== A-B =====")
res=np.polysub(A,B)

print(np.poly1d(res))

res=np.polysub(A,B)%q

print ("\n\nA-B (mod",q,"):")
print(np.poly1d(res))

print ("\n\nA/B =====")

res=np.floor(np.polydiv(A,B)[0])
```

```
print("Divide:\n",np.poly1d(res))

res=np.floor(np.polydiv(A,B)[1])
print("Remainder:\n",np.poly1d(res))

res=np.floor(np.polydiv(A,B)[0])%q
print ("\n\nA div B (mod",q,"):")
print(np.poly1d(res))


print("\n\nA div B (mod {0} ):".format(q))

print(np.poly1d(res))
```

A sample run using $A = 4x^3 + x^2 + 11x + 10$ and $B = 3x^3 + 5x^2 + 2x + 4$ is:

```
A:
   3     2
4 x + 1 x + 11 x + 10
B:
   3     2
3 x + 5 x + 2 x + 4


====== A+B =====
   3     2
7 x + 6 x + 13 x + 14

A+B (mod 13 ):
   3     2
7 x + 6 x + 1


====== A*B =====
    6      5      4       3      2
12 x + 23 x + 46 x + 103 x + 76 x + 64 x + 40


A*B (mod 13 ):
    6      5     4       3      2
12 x + 10 x + 7 x + 12 x + 11 x + 12 x + 1


====== A-B =====
   3     2
1 x - 4 x + 9 x + 6
```

```
A-B (mod 13 ):
   3      2
1 x + 9 x + 9 x + 6


A/B =====
Divide:

1
Remainder:
      2
-6 x + 8 x + 4


A div B (mod 13 ):

1
```

2.3.2 Inverse polynomial mod p

With lattice methods, we use the shortest vector problem in a lattice, which has an underpinning difficulty of factorizing polynomials. The lattice vector points are represented as polynomial values. For this, we create a modulo polynomial and then generate its inverse. In this case, we will create a polynomial (f) and then find its modulo inverse (f_p), and which will result in:

$$f \cdot f_p = 1 \pmod{p} \tag{2.45}$$

Thus, if we then take a message (m) and multiply it by f and then by f_p, we should be able to recover the message. Let's take an example with N=11 (the highest polynomial factor), p=31 and where Bob picks polynomial factors (f) of:

$$f = [-1, 1, 1, 0, -1, 0, 1, 0, 0, 1, -1] \tag{2.46}$$

$$f(x) = -1x^{10} + 1x^9 + 1x^6 - 1x^4 + 1x^2 + 1x - 1 \pmod{31} \tag{2.47}$$

We then determine the inverse of this with:

$$f_p : [9, 5, 16, 3, 15, 15, 22, 19, 18, 29, 5] \tag{2.48}$$

$$fp(x) = 5x^{10}+29x^9+18x^8+19x^7+22x^6+15x^5+15x^4+3x^3+16x^2+5x+9 \pmod{31} \tag{2.49}$$

So, we can now take a message (m) and multiply it by f to gain a ciphertext, and then multiply it with f_p to recover the message. In the following, we will use a message of:

$$m = [1, 0, 1, 0, 1, 1, 1, 1] \tag{2.50}$$

$$m = 1x^7 + 1x^6 + 1x^5 + 1x^4 + 1x^2 + 1 \tag{2.51}$$

A test run is [here]:

```
Values used:
 N= 11
 p= 31
========
Bob picks a polynomial (f):
f(x)=  [-1, 1, 1, 0, -1, 0, 1, 0, 0, 1, -1]
10      9      6      4      2
-1 x   + 1 x + 1 x - 1 x + 1 x + 1 x - 1
====Now we determine F_p ===
F_p: [9, 5, 16, 3, 15, 15, 22, 19, 18, 29, 5]
10       9       8       7       6       5       4       3       2
5 x   + 29 x + 18 x + 19 x + 22 x + 15 x + 15 x + 3 x + 16 x +
                                      5 x + 9
====Now we determine the message ===
Alice's Message:  [1, 0, 1, 0, 1, 1, 1, 1]
7      6      5      4      2
1 x + 1 x + 1 x + 1 x + 1 x + 1
====Encrypted message ===
Encrypted message:  [0, 1, 2, 1, 30, 0, 0, 1, 2, 1, 30]
10      9      8      7       4      3      2
30 x   + 1 x + 2 x + 1 x + 30 x + 1 x + 2 x + 1 x
====Decrypted message ===
Decrypted message:  [1, 0, 1, 0, 1, 1, 1, 1]
7      6      5      4      2
1 x + 1 x + 1 x + 1 x + 1 x + 1
```

The code is based on this code [42]:

```
from fracModulo import extEuclidPoly, modPoly, multPoly,
                                    reModulo
import numpy
import sys
```

```
N=11
p=31

f=[-1,1,1,0,-1,0,1,0,0,1,-1]
m=[1,0,1,0,1,1,1,1]

if (len(sys.argv)>1):
  N=int(sys.argv[1])
if (len(sys.argv)>2):
  p=int(sys.argv[2])
if (len(sys.argv)>3):
  f=eval("["+sys.argv[3]+"]")
if (len(sys.argv)>4):
  m=eval("["+sys.argv[4]+"]")



print("Values used:")
print(" N=",N)
print(" p=",p)
print("========")
print("\nBob picks a polynomials (f):")

print("f(x)= ",f)
print ("\n",numpy.poly1d(f[::-1]))

D=[0]*(N+1)
D[0]=-1
D[N]=1


print("\n====Now we determine F_p ===")
[gcd_f,s_f,t_f]=extEuclidPoly(f,D)

f_p=modPoly(s_f,p)

print("F_p:",f_p)
print ("\n",numpy.poly1d(f_p[::-1]))


x=multPoly(f,m)
enc=reModulo(x,D,p)

x=multPoly(enc,f_p)
dec=reModulo(x,D,p)[:len(m)]

print("\n====Now we determine F_p ===")
print("Alice's Message:\t",m)
print ("\n",numpy.poly1d(m[::-1]))
```

```
print("\n====Encrypted message ===")
print("Encrypted message:\t",enc)
print ("\n",numpy.poly1d(enc[::-1]))

print("\n====Decrypted message ===")
print("Decrypted message:\t",dec)
print ("\n",numpy.poly1d(dec[::-1]))
```

2.4 Learning With Errors

Learning With Errors (LWE) is a quantum robust method of cryptography. Initially we create a secret key value (s) and another value (e). Next we select a number of values (A[]) and calculate B[] = A[] x s + e. The values of A[] and B[] become our public key. If s is a single value, A and B are one dimensional matrices. If we select s to be a one-dimensional matrix, A will be a two-dimensional matrix, and B will be a one-dimensional matrix.

Learning with errors is a method defined by Oded Regev in 2005 [43] and is known as LWE (Learning With Errors). It involves the difficulty of finding the values which solve:

$$\mathbf{B} = \mathbf{A} \times \mathbf{s} + \mathbf{e} \tag{2.52}$$

where you know **A** and **B**. The value of **s** becomes the secret values (or the secret key), and **A** and **B** can become the public key A video is here [44].

2.4.1 Basics for LWE

In this presentation we will use the examples defined at [45]. The following code implements a basic LWE method:

```
import numpy as np
import sys

q=13

A=np.array([[4 ,1, 11, 10],[5, 5 ,9 ,5],[3, 9 ,0 ,10],[1, 3 ,
                                3 ,2],[12, 7 ,3 ,4],[6, 5 ,11
                                ,4],[3, 3, 5, 0]])

sA = np.array([[6],[9],[11],[11]])
eA = np.array([[0],[-1],[1],[1],[1],[0],[-1]])

bA = np.matmul(A,sA)\%q
print (bA)
```

```
bA = np.add(bA,eA)\%q
print
print ("Print output\n",bA)
```

$$b_A = \begin{bmatrix} 4 & 1 & 11 & 10 \\ 5 & 5 & 9 & 5 \\ 3 & 9 & 0 & 10 \\ 1 & 3 & 3 & 2 \\ 12 & 7 & 3 & 4 \\ 6 & 5 & 11 & 4 \\ 3 & 3 & 5 & 0 \end{bmatrix} \times \begin{bmatrix} 6 \\ 9 \\ 11 \\ 11 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} \pmod{13} = \begin{bmatrix} 4 \\ 7 \\ 2 \\ 11 \\ 5 \\ 12 \\ 8 \end{bmatrix}$$

2.4.2 Ring LWE

In this presentation we will use the examples defined at [45]. The following code implements a basic Ring LWE method:

```
import numpy as np


A = [4,1,11,10]
sA = [6,9,11,11]
eA =[0,-1,1,1]

n=len(A)
q=13

print (A,sA,eA)

xN_1 = [1] + [0] * (n-1) + [1]

print (xN_1)
A = np.floor(np.polydiv(A,xN_1)[1])

bA = np.polymul(A,sA)%q
bA = np.floor(np.polydiv(bA,xN_1)[1])

bA = np.polyadd(bA,eA)%q
bA = np.floor(np.polydiv(bA,xN_1)[1])
print ("Print output\n",bA)
```

In this method we perform a similar method to the Diffie Hellman method, but use Ring Learning With Errors (RLWE). With RLWE we use the coefficients of polynomials and which can be added and multiplied within a finite field ($\mathbf{F}_q$) [46] and where all the coefficients will be less than q. Initially Alice and Bob agree on a complexity value of n, and which is the highest

co-efficient power to be used. They then generate q which is $2^n - 1$. All the polynomial operations will then be conducted with a modulus of q and where the largest coefficient value will be $q - 1$. She then creates $a_i(x)$ which is a set of polynomial values:

$$\mathbf{A} = a_{n-1}x^{n-1} + ... + a_1x + a_1x^2 + a_0 \tag{2.53}$$

Next Alice will divide by $\Phi(x)$, which is $x^n + 1$:

$$\mathbf{A} = (a_{n-1}x^{n-1} + ... + a_1x + a_1x^2 + a_0) \div (x^n + 1) \tag{2.54}$$

In Python this is achieved with:

```
xN_1 = [1] + [0] * (n-1) + [1]
A = np.floor(p.polydiv(A,xN_1)[1])
```

$$e_A = e_{n-1}x^{n-1} + ...e_2x^2 + e_1x + e_0 \pmod q \tag{2.55}$$
$$s_A = s_{n-1}x^{n-1} + ...s_2x^2 + s_1x + s_0 \pmod q \tag{2.56}$$

Alice now creates b_A which is a polynomial created from A, e_A and s_A:

$$b_A = A \times s_A + e_A \tag{2.57}$$

In coding this is defined as:

```
bA = p.polymul(A,sA) \% q
bA = np.floor(p.polydiv(sA,xN_1)[1])
bA = p.polyadd(bA,eA) \% q
```

Bob now generates his own error polynomial $e^{'}$ and a secret polynomial $s^{'}$:

$$e_B = e^{'}_{n-1}x^{n-1} + ...e^{'}_2x^2 + e^{'}_1x + e^{'}_0 \pmod q \tag{2.58}$$
$$s_B = s^{'}_{n-1}x^{n-1} + ...s^{'}_2x^2 + s^{'}_1x + s^{'}_0 \pmod q \tag{2.59}$$

Alice shares A with Bob, and Bob now creates b_B which is a polynomial created from A, e_B and s_B:

$$b_B = A \times s_B + e_B \tag{2.60}$$

The code for this is:

```
sB = gen_poly(n,q)
eB = gen_poly(n,q)

bB = p.polymul(A,sB)\%q
bB = np.floor(p.polydiv(sB,xN_1)[1])
bB = p.polyadd(bB,eB)\%q
```

Alice then takes Bob's value (b_B) and multiplies by s_A and divides by $x^n + 1$:

$$shared_A = b_B \times s_A \div (x^n + 1) \tag{2.61}$$

Bob then takes Alice's value (b_A) and multiplies by s_B and divides by $x^n + 1$:

$$shared_B = b_A \times s_B \div (x^n + 1) \tag{2.62}$$

At the end of this, Bob and Alice will have the same shared value (and thus can create a shared key based on it).

In code this is:

```
sharedAlice = np.floor(p.polymul(sA,bB) \%q)
sharedAlice = np.floor(p.polydiv(sharedAlice,xN_1)[1])\%q
sharedBob = np.floor(p.polymul(sB,bA)\%q)
sharedBob = np.floor(p.polydiv(sharedBob,xN_1)[1])\%q
```

Chapter 3

State-of-the-art

3.1 Introduction

Homomorphic encryption supports mathematical operations on encrypted data. In 1978, Rivest, Adleman, and Dertouzos [47] were the first to define the possibilities of implementing a homomorphic operation and used the RSA method. This supported multiply and divide operations [48], but does not support addition and subtraction. Overall, Partial Homomorphic Encryption (PHE) supports a few arithmetic operations, while Fully Homomorphic Encryption (FHE) supports add, subtract, multiply, and divide.

Since, Gentry defined the first FHE method [49] in 2009, there have been four main generations of homomorphic encryption:

- 1st generation: Gentry's method uses integers and lattices [50] including the DGHV method.
- 2nd generation. Brakerski, Gentry and Vaikuntanathan's (BGV) and Brakerski/Fan-Vercauteren (BFV) use a Ring Learning With Errors approach [51]. The methods are similar to each other, and with only small difference between them.
- 3rd generation: These include DM (also known as FHEW) and CGGI (also known as TFHE) and support the integration of Boolean circuits for small integers.
- 4th generation: CKKS (Cheon, Kim, Kim, Song) and which uses floating-point numbers [52].

Generally, CKKS works best for real number computations and can be applied to machine learning applications as it can implement logistic regression

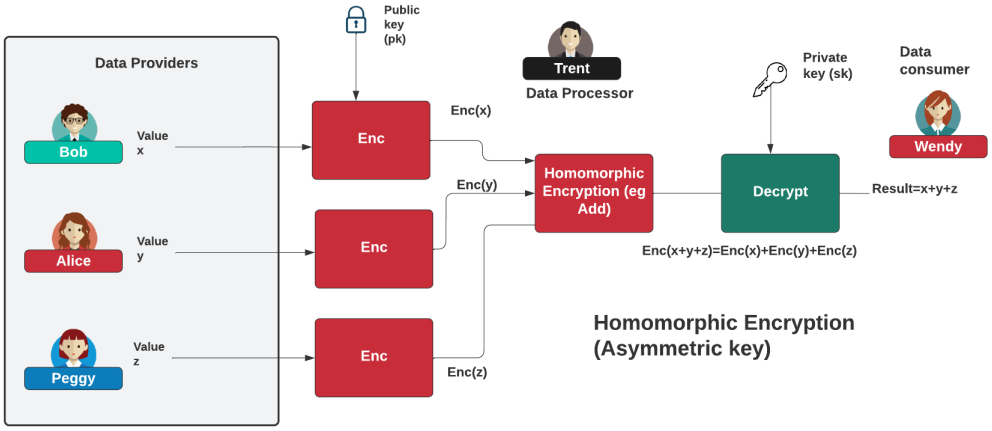

Figure 3.1: Asymmetric encryption (public key)

methods and other statistical computations. DM (also known as FHEW) and CGGI (also known as TFHE) are useful in the application of Boolean circuits for small integers. BGV and BFV are generally used in applications with small integer values.

3.2 Public key or symmetric key

Homomorphic encryption can be implemented either with a symmetric key or an asymmetric (public) key. With symmetric key encryption, we use the same key to encrypt as we do to decrypt, whereas, with an asymmetric method, we use a public key to encrypt and a private key to decrypt. In Figure 4.1 we use asymmetric encryption with a public key (pk) and a private key (sk). With this Bob, Alice and Peggy will encrypt their data using the public key to produce ciphertext, and then we can operate on the ciphertext using arithmetic operations. The result can then be revealed by decrypting with the associated private key. In Figure 4.2 we use symmetric key encryption, and where the data is encrypted with a secret key, and which is then used to decrypt the data. In this case, the data processor (Trent) should not have access to the secret key, as they could decrypt the data from the providers.

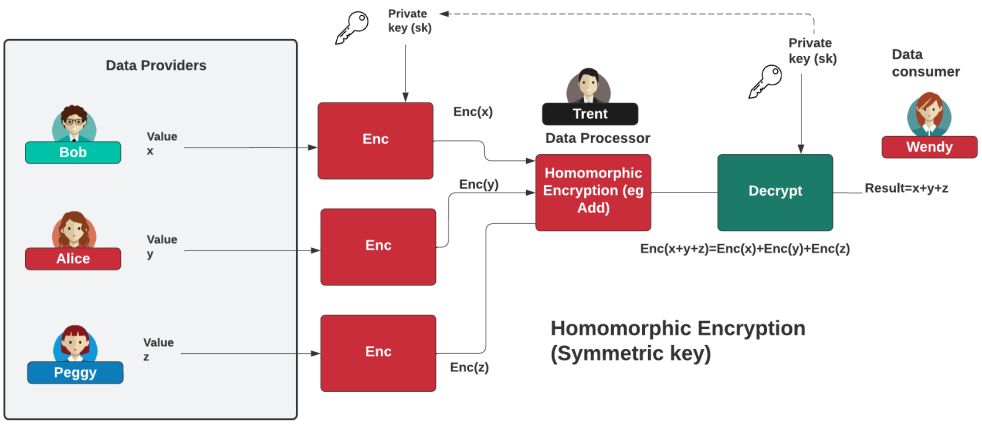

Figure 3.2: Symmetric encryption

3.3 Homomorphic libraries

There are several homomorphic encryption libraries that support FHe, including ones which support CUDA and GPU acceleration, but many have not been kept up-to-date with modern methods, or only integrate one method. Overall, the native language libraries tend to be the most useful, as they allow the compilation to machine code. The main languages for this are C++, Golang and Rust, although some Python libraries exist through wrappers to C++ code. This includes HEAAN-Python, and its associated HEAAN library.

One of the first libraries which supported a range of methods are Microsoft SEAL [53], SEAL-C# [54] and SEAL-Python. While it supports a wide range of methods, including BGV/BFV and CKKS, it has lacked any real serious development for the past few years. It does have support for Android and has a Node.js port [55]. One of the most extensive libraries is PALISADE, and which has now developed into OpenFHE. Within OpenFHE, the main implementations are:

- Brakerski/Fan-Vercauteren (**BFV**) scheme for integer arithmetic
- Brakerski-Gentry-Vaikuntanathan (**BGV**) scheme for integer arithmetic
- Cheon-Kim-Kim-Song (**CKKS**) scheme for real-number arithmetic (includes approximate bootstrapping)
- Ducas-Micciancio (**DM**) and Chillotti-Gama-Georgieva-Izabachene (**CGGI**) schemes for Boolean circuit evaluation.

Product	Creator	Language	License	Summary
SEAL [177]	Microsoft	C++	MIT	Widely-used FHE library that implements BFV for modular arithmetic and CKKS for approximate arithmetic.
HElib [116]	IBM	C++	Apache-2.0	Widely-used FHE library that implements BGV for modular arithmetic and CKKS for approximate arithmetic.
TFHE [59]	Gama et al.	C++	Apache-2.0	Implements an optimized ring variant of the GSW scheme.
HEAAN [115]	CryptoLab, Inc.	C++	CC-BY-NC-3.0	Implements the CKKS approximate number arithmetic scheme.
PALISADE [165]	New Jersey Institute of Technology	C++	BSD-2-Clause	Lattice cryptography library that supports multiple protocols for FHE, including BGV, BFV, and StSt.
$\Lambda \circ \lambda$ [69]	E. Crockett and C. Peikert	Haskell	GPL-3.0-only	Pronounced "LOL." Implements a BGV-type FHE scheme.
Cingulata [45]	CEA LIST	C++	CECILL-1.0	Compiler and RTE for C++ FHE programs. Implements BFV and supports TFHE.
FV-NFLlib [89]	CryptoExperts	C++	GPL-3.0-only	Implements FV scheme. Built on the NFLLib lattice cryptography library. Last updated 2016.
Lattigo [138]	Laboratory for Data Security	Go	Apache 2.0	Implements BFV and HEAAN in Go.

Figure 3.3: Homomorphic libraries [56]

Wood et al. [56] define a full range of libraries (Figure 3.3).

3.4 Bootstrapping

A key topic within fully homomorphic encryption is the usage of bootstrapping. Within a learning with-errors approach, we add noise to our computations. For a normal decryption process, we use the public key to encrypt data and then the associated private key to decrypt it. Within the bootstrap version of homomorphic encryption, we use an encrypted version of the private key that operates on the ciphertext. In this way, we remove the noise which can build up in the computation. Figure 3.4 outlines that we perform and evaluation on the decryption using an encrypted version of the private key. This will remove noise in the ciphertext, after which we can then use the actual private key to perform the decryption.

The main bootstrapping methods are CKKS [52], DM [57]/CGGI, and BGV/BFV. Overall, CKKS is generally the fastest bootstrapping method, while DM/CGGI is efficient with the evaluation of arbitrary functions. These functions approximate maths functions as polynomials (such as with Chebyshev approximation). BGV/BFV provides reasonable performance and is

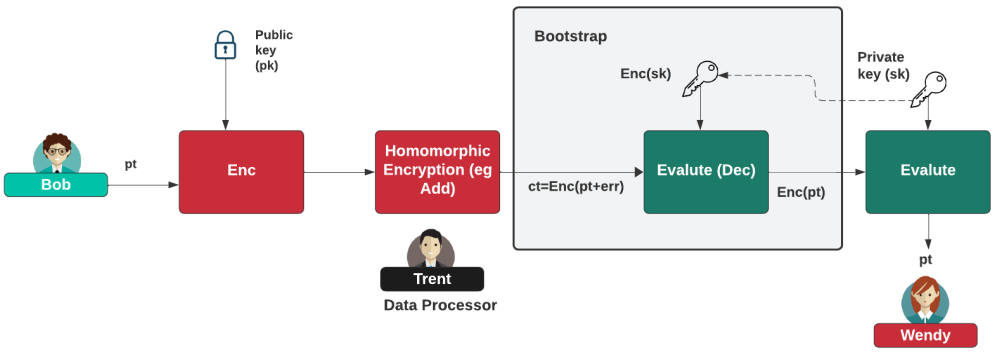

Figure 3.4: Bootstrap

generally faster than DM/CGGI but slower than CKKS.

3.5 Arbitrary smooth functions

With approximation theory, it is possible to determine an approximate polynomial $p(x)$ that is approximate to a function $f(x)$. A polynomial takes the form of $p(x) = a_n.x^n + a_{n-1}.x^{n-1} + ... + a_1.x + a_0$, and where $a_0 ... \ a_n$ are the coefficients of the powers, and n is the maximum power of the polynomial.

For this, we can define arbitrary smooth functions for CKKS using Chebyshev approximation [58]. These were initially created by Pafnuty Lvovich Chebyshev. This method involves the approximation of a smooth function using polynomials. Examples of these functions include log_{10}, log_2, log_e, and e^x [59].

3.6 Plaintext slots

With many homomorphic methods, we can encrypt multiple plaintext values into ciphertext in a single operation. This is defined as the number of plaintext slots, and is illustrated in Figure 3.5.

3.7 DGHV

Craig Gentry was the first to show an implementation of fully homomorphic encryption (FHE), and received the 2009 ACM Doctoral Dissertation Award for this breakthrough work. The concept of homomorphic encryption,

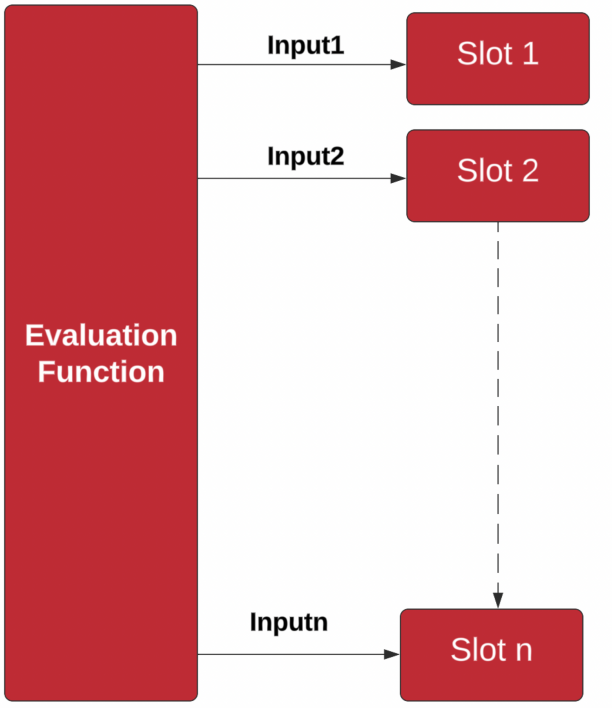

Figure 3.5: Slots for plaintext

though, had been around since public key encryption itself, and partial homomorphic methods included Paillier [37] and RSA [40]. Overall, homomorphic encryption can either be partially homomorphic (PHE), somewhat homomorphic, levelled fully homomorphic, or **fully homomorphic encryption** (FHE). For security, the best case is fully homomorphic encryption. Craig's published work on homomorphic encryption started in 2019 with the first useable fully homomorphic encryption (FHE) method [60].

The work of Gentry was then advanced, and, in 2010, DGHV (Dijk, Gentry, Halevi and Vaikuntanathan) defined an expansion into the usage of integers [50]. With these methods, we encrypt our data and then input these cipher bits into a cipher circuit. A basic adding function is here [61]. Overall it was not possible to examine any of the bits in the circuit to discover the original state of the plaintext. On the output of the circuit, we could then decrypt the cipher bits back into the cipher text. For this, we could create the function or arithmetic operation that we wanted ... it all comes down to EX-OR (adding) and AND (multiplication) operations.

3.8 BGV and BFV

With BGV and BFV, we use a Ring Learning With Errors (LWE) method [51]. With BGV, we define a moduli (q), which constrains the range of the polynomial coefficients. Overall, the methods use a moduli, which can be defined within different levels. We then initially define a finite group of $\mathbb{Z}_q$, and then make this a ring by dividing our operations with $(x^n + 1)$ and where $n-1$ is the largest power of the coefficients. The message can then be represented in binary as:

$$m = a_{n-1}a_{n-2}...a_0 \tag{3.1}$$

This can be converted into a polynomial with:

$$\mathbf{m} = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0 \pmod{q} \tag{3.2}$$

The coefficients of this polynomial will then be a vector. Note that for efficiency, we can also encode the message with ternary (such as with -1, 0 and 1). We then define the plaintext modulus with:

$$t = p^r \tag{3.3}$$

and where p is a prime number and r is a positive number. We can then define a ciphertext modulus of q, and which should be much larger than t. To encrypt with the private key of $\mathbf{s}$, we implement:

$$(c_0, c_1) = \left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e, -\mathbf{a}\right) \mod q \tag{3.4}$$

To decrypt:

$$m = \lfloor \frac{t}{q}(c_0 + c_1).\mathbf{s} \rceil \tag{3.5}$$

This works because:

$$m_{recover} = \lfloor \frac{t}{q}\left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e - \mathbf{a}.\mathbf{s}\right) \rceil \tag{3.6}$$

$$= \lfloor \left(\mathbf{m} + \frac{t}{q}.e\right) \rceil \tag{3.7}$$

$$\approx m \tag{3.8}$$

$$\tag{3.9}$$

For two message of m_1 and m_2, we will get:

$$Enc(m_1 + m_2) = Enc(m_1) + Enc(m_2) \tag{3.10}$$

$$Enc(m_1.m_2) = Enc(m_1).Enc(m_2) \tag{3.11}$$

3.8.1 Noise and computation

But each time we add or multiply, the error also increases. Thus bootstrapping is required to reduce the noise. Overall, addition and plaintext/ciphertext multiplication is not a time-consuming task, but ciphertext/ciphertext multiplication is more computationally intensive. The most computational task is typically the bootstrapping process, and the ciphertext/ciphertext multiplication process adds the most noise to the process.

3.8.2 Parameters

We thus have a parameter of the ciphertext modulus (q) and the plaintext modulus (t). Both of these are typically to the power of 2. An example of q is 2^{240} and for t is 65,537. As the value of 2^q is likely to be a large number, we typically define it as a log_q value. Thus a ciphertext modulus of 2^{240} will be 240 as defined as a log_q value.

3.8.3 Public key generation

We select the private (secret) key using a random ternary polynomial (-1, 0, and 0 coefficients) and which has the same degree as our ring. The public key is then a pair of polynomials as:

$$\mathbf{pk_1} = (\mathbf{r}.\mathbf{sk} + e) \pmod{q} \tag{3.12}$$

$$\mathbf{pk_2} = \mathbf{r} \tag{3.13}$$

and where r is a random polynomial value. To encrypt with the public key (pk):

$$(c_0, c_1) = \left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s}.\mathbf{r} + e, -\mathbf{a}.\mathbf{r}\right) \mod q \tag{3.14}$$

We then decrypt with the private key (s);

$$m = \lfloor \frac{t}{q}(c_0 + c_1).\mathbf{s} \rceil \tag{3.15}$$

This works because:

$$m_{recovered} = \lfloor \frac{t}{q}\left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s}.\mathbf{r} + e - \mathbf{a}.\mathbf{r}.\mathbf{s}.\right) \rceil \tag{3.16}$$

$$= \lfloor \left(\mathbf{m} + \frac{t}{q}.e\right) \rceil \tag{3.17}$$

$$\approx m \tag{3.18}$$

$$\tag{3.19}$$

Coding examples

The following are coding examples:

- Adding Two Numbers with Homomorphic Encryption for BGV and BFV using OpenFHE and C++. Coding: Here
- Multiplying Two Numbers with Homomorphic Encryption for BGV and BFV using OpenFHE and C++. Coding: Here
- Batch Encoding with BFV/BGV using OpenFHE and C++ (Squaring). Coding: Here

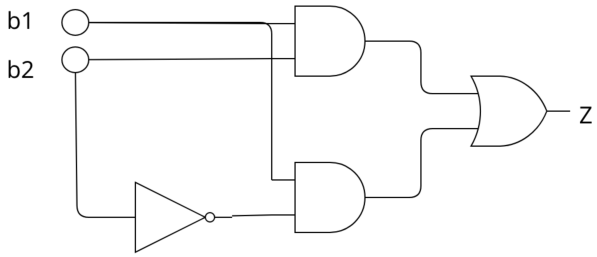

Figure 3.6: Boolean circuit

3.9 DM(FHEW) and CGGI(TFHE)

The work in the 3rd generation of the method outlined the FHEW scheme [62] that allowed for the refreshing of ciphertexts after every operation. This reduced the bootstrapping time to less than a second. It can be used with the bootstrapping technique proposed by Ducas and Micciancio (DM) [57]. Overall, FHEW uses Boolean gates applied to cipher data bits, and which simplifies the bootstrapping process. An enhancement for DM(FHEW) is to use the CGGI(TFHE) schemes [63].

Overall the standard gates that we implement are AND/NAND, OR/NOR, XON/XNOR, MUX and Majority. The input to the gates are cipher bits. An example of coding with Figure 3.6 is [64]:

```
auto bit1 = cc.Encrypt(sk, b1);
auto bit2 = cc.Encrypt(sk, b2);

auto ctAND1 = cc.EvalBinGate(AND, bit1, bit2);
auto bit2Not = cc.EvalNOT(bit2);
auto ctAND2 = cc.EvalBinGate(AND, bit2Not, bit1);
auto ctResult = cc.EvalBinGate(OR, ctAND1, ctAND2);
```

3.9.1 DM

DM (FHEW) [57] uses an LWE (Learning With Error) method that provides fully homomorphic encryption (FHE). It is able to evaluate cipher data applied onto Boolean circuits and uses bootstrapping after each gate evaluation. This allows the evaluation to be conducted in less than 0.1 seconds. The gates implemented are AND, OR, NAND, NOR, and NOT. In homomorphic encryption, the noise in the computation increases as we perform operations. This is typically when we perform a large number of additions. and multiplications will often reduce the amount of noise. A bootstrapping

process allows us to reset the noise and will introduce a delay in the computation. The OpenFHE library supports the usage of the GINX bootstrapping method [65], AP [66], or LMKCDEY [67].

Coding examples

The following are coding examples:

- Boolean Circuits with OpenFHE using the FHEW and bootstrapping method (GINX). Coding: Here.
- Boolean Circuit Evaluation using the DM (FHEW) with LMKCDEY bootstrapping and OpenFHE. Boolean Circuit Evaluation using the DM (FHEW) with LMKCDEY bootstrapping and OpenFHE. Coding: Here
- Boolean Circuits with OpenFHE using the FHEW and bootstrapping method (GINX). Boolean Circuits with OpenFHE using the FHEW and bootstrapping method (GINX) with Public Key Encryption. DM (FHEW) [1] uses a LWE (Learning With Error) method that provides fully homomorphic encryption (FHE). Coding: Here.
- Millionaire's Problem: Simple Homomorphic Cipher for proving if Bob is older than Alice with MD (FHEW). Simple Homomorphic Cipher for proving if Bob is older than Alice with MD (FHEW). Coding: Here.
- Millionaire's Problem: Simple Homomorphic Cipher for proving if Bob is older than Alice with MD (FHEW) with PKE. Simple Homomorphic Cipher for proving if Bob is older than Alice with MD (FHEW) with PKE. Coding: Here.
- Homomorphic Cipher - 4-bit adder using MD (FHEW). Homomorphic Cipher - 4-bit adder using MD (FHEW). This is a simple abstraction of Homomorphic Cipher - 4-bit adder. Coding: Here.
- Truth Table with Cipher Circuit with MD (FHEW) with PKE. Coding: Here.
- Majority and MUX Gates with OpenFHE using the DM (FHEW) and bootstrapping (GINX) using PKE. Majority and MUX Gates with OpenFHE using the DM (FHEW) and bootstrapping (GINX) using PKE. Coding: Here.

- Increasing Plaintext Range for Binary Evaluation with OpenFHE and C++. Coding: Here.
- FHEW scheme for a small precision arbitrary function evaluation. Coding: Here.
- Gray Code using the DM (FHEW) and OpenFHE using Symmetric Key Encryption. Coding: Here.

3.10 CKKS

HEAAN (Homomorphic Encryption for Arithmetic of Approximate Numbers) defines a homomorphic encryption (HE) library proposed by Cheon, Kim, Kim and Song (CKKS). The CKKS method uses approximate arithmetics over complex numbers [52]. Overall, it is a levelled approach that involves the evaluation of arbitrary circuits of bounded (pre-determined) depth. These circuits can include ADD (X-OR) and Multiply (AND).

HEAAN uses a rescaling procedure to measure the size of the plaintext. It then produces an approximate rounding due to the truncation of the ciphertext into a smaller modulus. The method is especially useful in that it can be applied to carry out encryption computations in parallel. Unfortunately, the ciphertext modulus can become too small, and where it is not possible to carry out any more operations.

The HEAAN (CKKS) method uses approximate arithmetic over complex numbers ($\mathbb{C}$) and is based on Ring Learning With Errors (RLWE). It focuses on defining an encryption error within the computational error that will happen within approximate computations. We initially take a message (M) and convert it to a cipher message (ct) using a secret key sk. To decrypt ($[\langle ct,sk\rangle]q$), we produce an approximate value along with a small error (e).

Craig Gentry [60] has outlined three important application areas within privacy-preserving genome association, neural networks, and private information retrieval. Along with this, he proposed that the research community should investigate new methods which **did not involve the usage of lattices**.

3.10.1 Coding examples

The following are coding examples:

- Add, Subtract and Multiplying Two Numbers with Homomorphic Encryption for CKKS using OpenFHE and C++. Coding: Here

3.10.2 Chebyshev approximation

With approximation theory, it is possible to determine an approximate polynomial $p(x)$ that is approximate to a function

$$(x)$$

. A polynomial takes the form form of $p(x) = a_n.x^n + a_{n-1}.x^{n-1} + a_1.x + a_0$, and where $a_0...a_n$ are the coefficients of the powers, and n is the maximum power of the polynomial. In this case, we will evaluate arbitrary smooth functions for CKKS and using Chebyshev approximation. These were initially created by Pafnuty Lvovich Chebyshev. This method involves the approximation of a smooth function using polynomials.

Overall, with polynomials, we convert our binary values into a polynomial, such as 101101 is:

$$x^5 + x^3 + x^2 + 1 \tag{3.20}$$

Our plaintext and ciphertext are then represented as polynomial values.

Approximation theory

With approximation theory, we aim to determine an approximate method for a function f(x). It was Pafnuty Lvovich Chebyshev who defined a method of finding a polynomial p(x) that is approximate for f(x). Overall, a polynomial takes the form of:

$$p(x) = a_n.x^n + a_{n-1}.x^{n-1} + a_1.x + a_0 \tag{3.21}$$

and where $a_0...a_n$ are the coefficients of the powers, and n is the maximum power of the polynomial. Chebyshev published his work in 1853 as "Theorie des mecanismes, connus sous le nom de parall´elogrammes". His problem statement was "to determine the deviations which one has to add to get an approximated value for a function f, given by its expansion in powers of x-a, if one wants to minimise the maximum of these errors between x = a - h and x = a + h, h being an arbitrarily small quantity".

Let's find the polynomial that finds the magnitude of x: f(x)=—x— for an interval of -1 to 1. For this, we can use:

If we try x=0.5, we get:

$$p(0.5) = 15/16.0.5 + 3/16 = 0.421875 \tag{3.22}$$

and for x=-0.5, we get:

$$p(-0.5) = 15/16.(-0.5) + 3/16 = 0.421875 \tag{3.23}$$

Mathematically we can prove that the maximum error will be 0.1021. Overall, we can use a Chebyshev polynomial of degree n by $T_n(x)$ as:

$$T_n(x) = cos(n.cos^{-1}(x)) \tag{3.24}$$

For $n = 0$, we get:

$$T_0(x) = cos(0.cos^{-1}(x)) = 1 \tag{3.25}$$

For $n = 1$, we get:

$$T_1(x) = cos(1.cos^{-1}(x)) = x \tag{3.26}$$

Coding examples

- Chebyshev approximations using OpenFHE and C++ (Logarithm methods). Coding: Here

3.10.3 Polynomial evaluations

A polynomial takes the form form of $p(x) = a_n.x^n + a_{n-1}.x^{n-1} + a_1.x + a_0$, and where $a_0...a_n$ are the coefficients of the powers, and n is the maximum power of the polynomial. With CKKS in OpenFHE, we can evaluate the result of a polynomial for a given range of x values. For example, if we have $p(x) = 5.x^2 + 3.x + 7$ will give a result of $p(2) = 33$.

Code examples

- Polynomial Evaluation using OpenFHE and C++. Coding: Here

3.11 Conclusions

For the state-of-the-art, the best library for support seems to be OpenFHE, as it supports BGV/BFV, CKKS and DM/CGGI, along with other cryptographic primitives, such as proxy re-encryption and secret shares. Overall, CKKS is the fastest method for homomorphic encryption and supports floating-point values. The DM/CGGI method supports Boolean logic circuits but can be slow in the computation. For performance, the bootstrapping processing and ciphertext-ciphertext multiplication have the highest performance impact.

Chapter 4

Data Sharing

4.1 Introduction

Encryption can exist in three states: in transit, in memory and at rest. Unfortunately our traditional encryption methods tend to have to decrypt data in these states so that it can be processed. With homomorphic encryption, we can encrypt data at the source and then use it to process the data.

4.2 Overview

As previously defined, we can either use public key encryption (Figure 4.1) or symmetric key encryption (Figure 4.2) to perform homomorphic encryption and decryption. In most cases, we will be using public key encryption, as it can be difficult to protect the private key ‘(unless it is stored in a secure enclave. The most basic data-sharing example is where Bob, Alice and Wendy encrypt data from its source and then feed this into a homomorphic engine. The result can then be decrypted by whoever has the associated private key. A key note is that data encrypted with a certain public key cannot be used in another processing operation and can only be decrypted with the associated private key. This allows for a separation of the data, and where a data owner may give consent for a given key pair to be used.

4.3 Applications in Data Sharing

Munjal et al. [68] outline a healthcare scenario with homomorphic encryption. With this, a patient encrypts data with the homomorphic public key, and then uploads this to the Cloud. This data can then be computed using

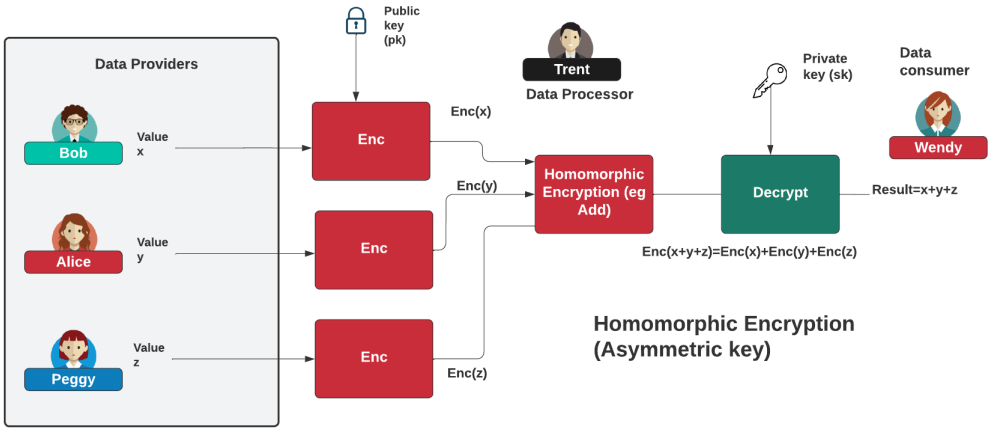

Figure 4.1: Asymmetric encryption (public key)

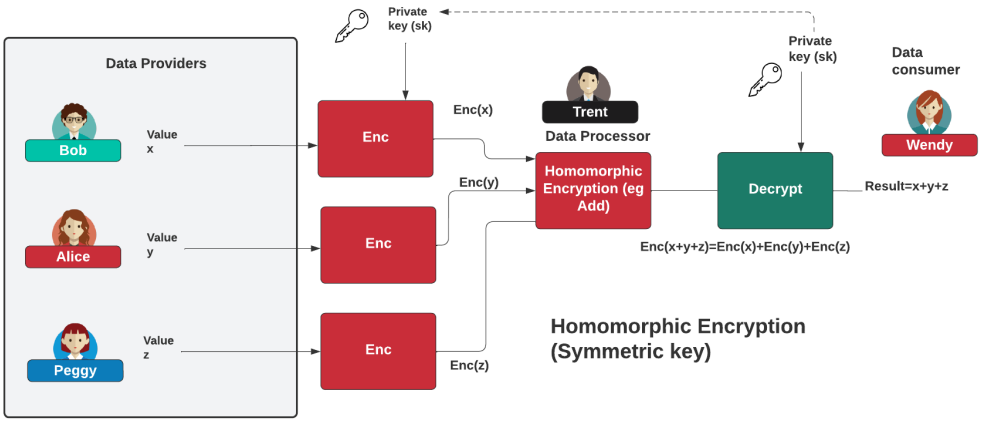

Figure 4.2: Symmetric encryption

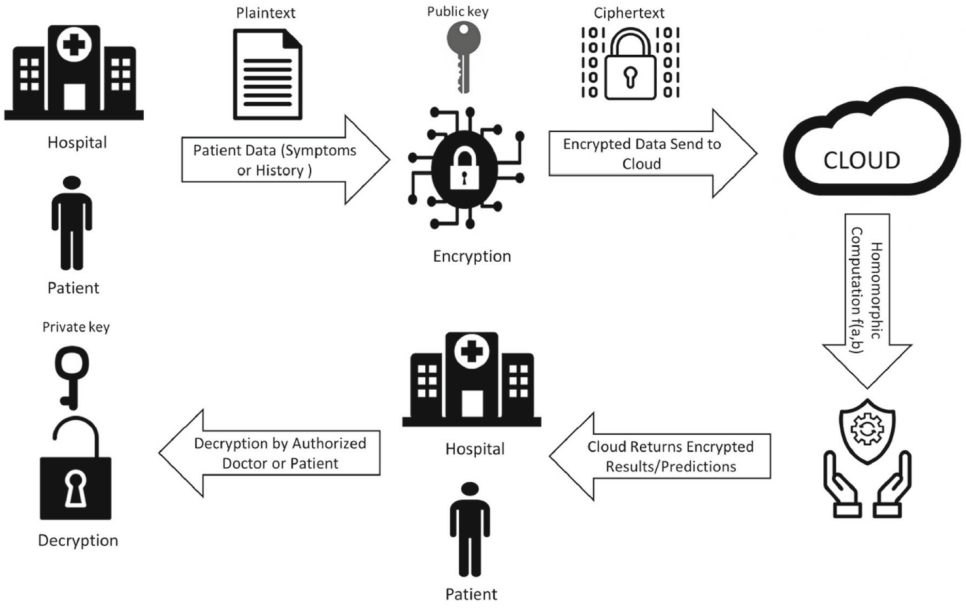

Figure 4.3: Homomorphic encryption for data sharing [68]

homomorphic encryption methods and decrypted by the patient or related clinician. One of the application areas is in ECG Monitoring [69].

Geva et [70] have set up oncological data using multiparty homomorphic encryption. As shown in Figure 4.4, each of the collaborators has a share of the key. These are generated from a distributed key generation approach. The data owner (DO) then encrypts their data with a well-known public key. The computation is then done by the parties with their key shares, and the computation is then aggregated and sent back to the DO for them to decrypt with their private key.

4.4 Symmetric-key encryption methods

Halder et al. [71] outline SmartCrypt, which defines a homomorphic encryption-based flexible and fine-grain sharing technique (Figure 4.5. It uses a symmetric key encryption approach and supports the computation of sum, mean and count. This method is TinyEnc [72] and is an ordering encryption method.

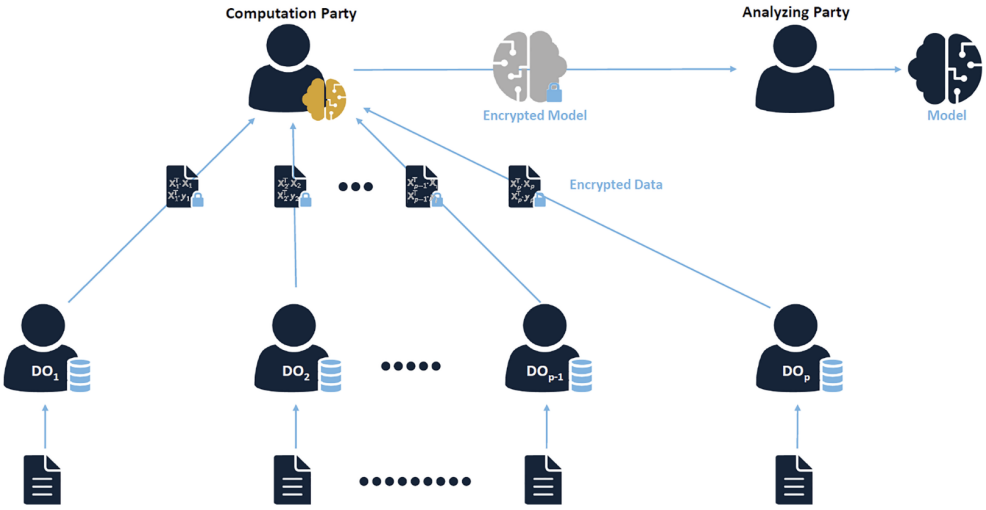

Figure 4.4: Multiparty sharing with homomorphic encryption [70]

4.4.1 Order-preserving encryption (OPE)

Rahman et al. outline an Order-preserving encryption (OPE) for encrypted searches using symmetric key encryption. [73]. Overall, OPE is a symmetric key encryption method that can be used to create an order on encrypted content. We will lose some information in supporting the ordering of the ciphers, but the security level should still stay acceptable for the application area. If we encrypt a value of A with a key of k to get $E_k(A)$, and then encrypt B to get $E_k(B)$. Then if $A < B$, then:

$$Ek(A) < Ek(B) \tag{4.1}$$

One of the best papers on the subject is [74], and an article is [75].

4.5 Federated HE

Shi et al [76] outline a federated system which uses Paillier homomorphic encryption. With this, they define a number of key comparisons:

1. Parameter confidentiality.
2. Resistance to collusion.
3. Non client-interactive.

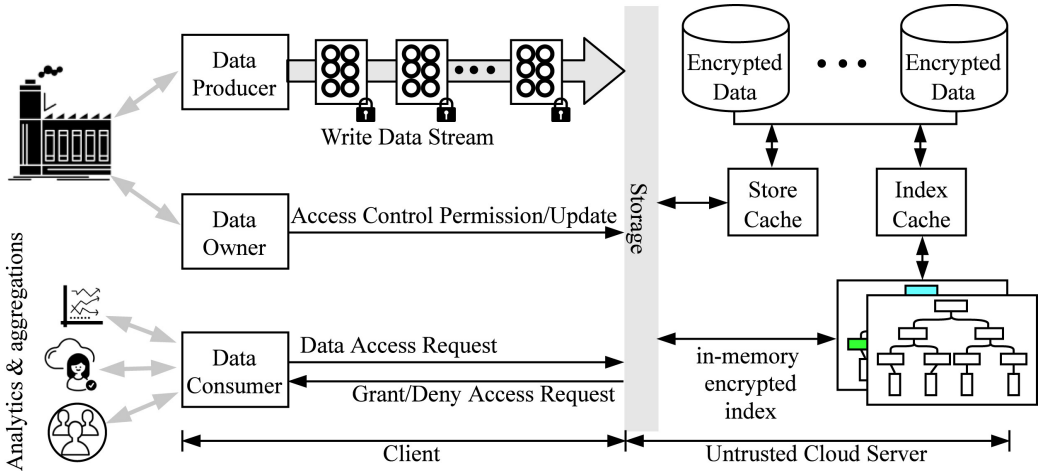

Figure 4.5: SmartCrypt[71]

4. Without uniform private key.

5. Communication rounds.

4.6 Proxy re-encryption

With public key encryption, we can encrypt with a public key and then decrypt with a private key. And, so, if we have some ciphertext, and we want to pass this on to Bob, we would have to re-encrypt by decrypting with Alice's private key and then re-encrypt with Bob's public key. With proxy re-encryption (PRE), we can convert ciphertext encrypted with Alice's public key into one that Bob can decrypt with his private key. as shown in Figure 4.6. This feature is supported in OpenFHE [77].

In this case, we might have a trusted proxy agent, such as Trent, and who is trusted to use Alice's private key. In this case, we might have a trusted proxy agent, such as Trent, and who is trusted to use Alice's private key. Trent can either be trusted to hold Alice's private key, or Alice can use her private key and Bob's public key to create the re-encryption key.

4.6.1 Coding examples

A practical coding example is:

- Proxy Re-encryption (PRE) with CKKS Homomorphic Encryption using OpenFHE and C++. Proxy Re-encryption (PRE) with CKKS

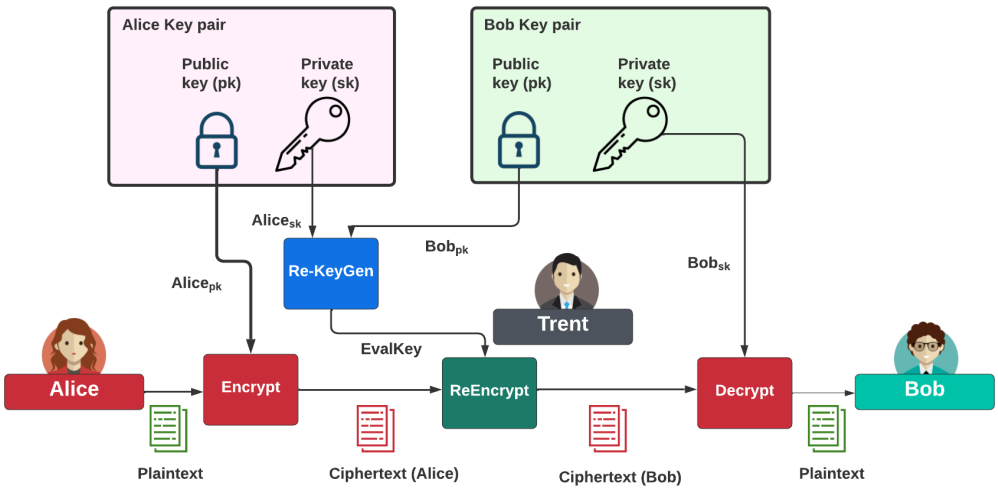

Figure 4.6: Proxy re-encryption

Homomorphic Encryption using OpenFHE and C++. With public key encryption, we can encrypt with a public key and then decrypt with a private key. And, so, if we have some ciphertext, and we want to pass this on to Bob, we would have to re-encrypt by decrypting with Alice's private key and then re-encrypt with Bob's public key. With proxy re-encryption (PRE), we can convert ciphertext encrypted with Alice's public key into one that Bob can decrypt with his private key. Coding: Here.

Chapter 5

Cloud

5.1 Introduction

The usage of homomorphic encryption is well set up for cloud-based systems, where there is no need for a trusted entity to perform processing. Overall, the main options include using:

- Partially homomorphic encryption. This includes the usage of the Paillier method encryption for addition, subtraction and scalar multiplication [78].
- Full Homomorphic encryption. This includes the methods of BGV, CKKS and DM [79].

Applications of cloud-based systems which use homomorphic encryption, including privacy-aware location tracking, privacy-aware machine learning, privacy-aware biometric/ID matching, and privacy-aware searchable encryption.

5.2 Symmetric key

For homomorphic encryption, we can either use symmetric key or asymmetric key encryption. With a symmetric key, we use the same key to encrypt and decrypt. In a cloud-based system, we would have to make sure that the data was encrypted outside the cloud and then the decryption also outside. For this, the symmetric key needs to be stored in a secure enclave for the decryption process. The key must then be wrapped and loaded onto the devices that encrypt the data (Figure 5.1).

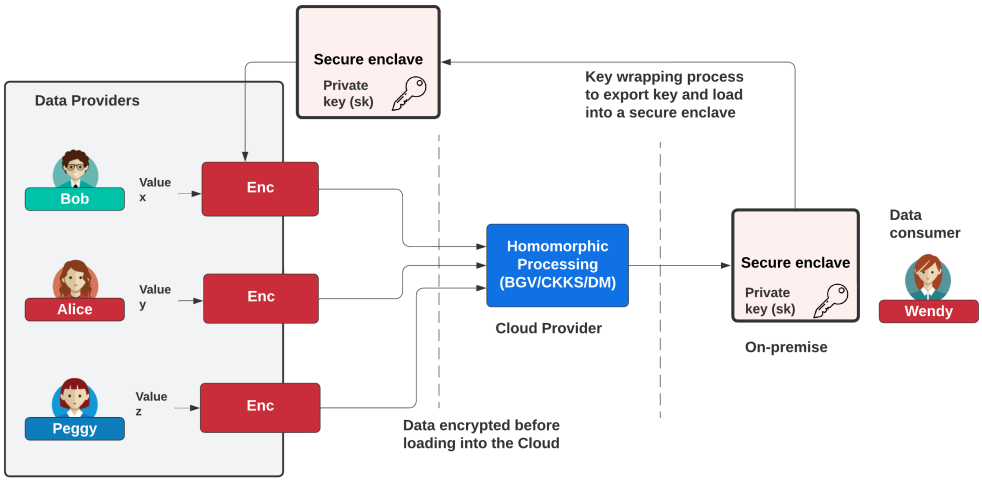

Figure 5.1: Symmetric key encryption in the cloud

5.3 Asymmetric key

With asymmetric key encryption, we can generate a homomorphic key pair where the private key can be kept in a secure enclave. We can then export the associated public key that can be used to encrypt data. The data passed to the cloud will thus be encrypted with the public key, and where the cloud provider will not have access to the private key. There are thus no security issues in handling the data in the cloud. Once processed, the data can be passed to a secure enclave and then decrypted by the secure enclave (Figure 5.2).

5.4 Searchable Encryption in the Cloud

Amorim et al [80] defined a homomorphic encryption model for a cloud search system (Figure 5.3, and for the key characteristics for a cloud-based Searchable Encryption method (Figure 5.4). Within searchable encryption we can have a keyword-value pair, and where the keyword can be a homomorphically encrypted value, and where the value can also be encrypted. We can then look for a match between an encrypted search keyword and the encrypted keyword on a database. The characteristics include:

- Search Structure. With a simple index we use keywords that relate to a document. For an inverted index, we have an index term that relates

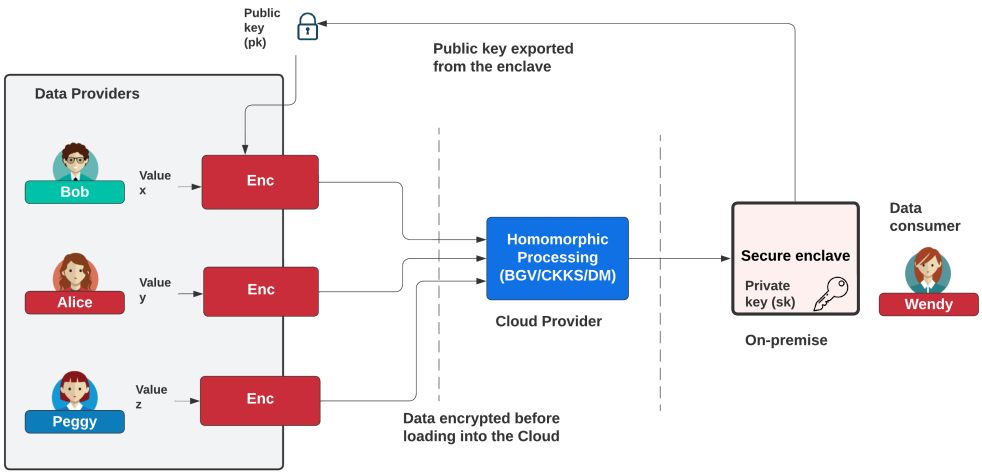

Figure 5.2: Symmetric key encryption in the cloud

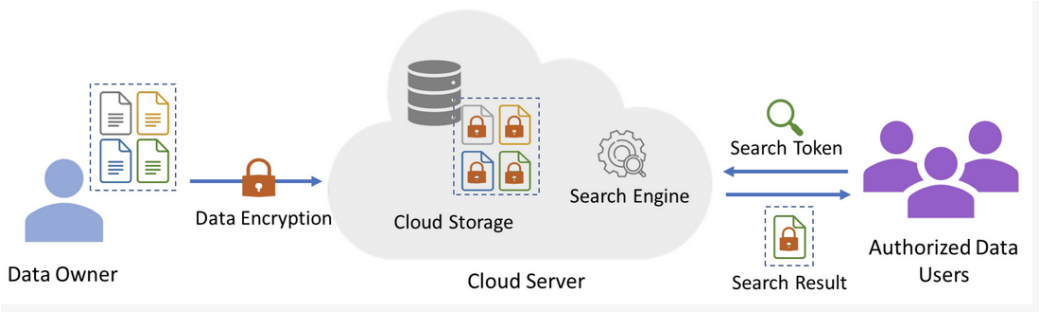

Figure 5.3: A conceptual overview of a cloud-based SE system [80]

to a number of documents where it appears. With a tree index, we have a tree-like structure for searchable keywords. With this, we divide keys into smaller sets, and where we can traverse through the tree until a matching node is found.

- Search Functionalities. For this we can have single keywords or multiple ones. The methods that can be employed for the searches include wildcards and regular expressions, fuzzy keyword search, phrase search, range search, occurrence search, and ranked search.
-

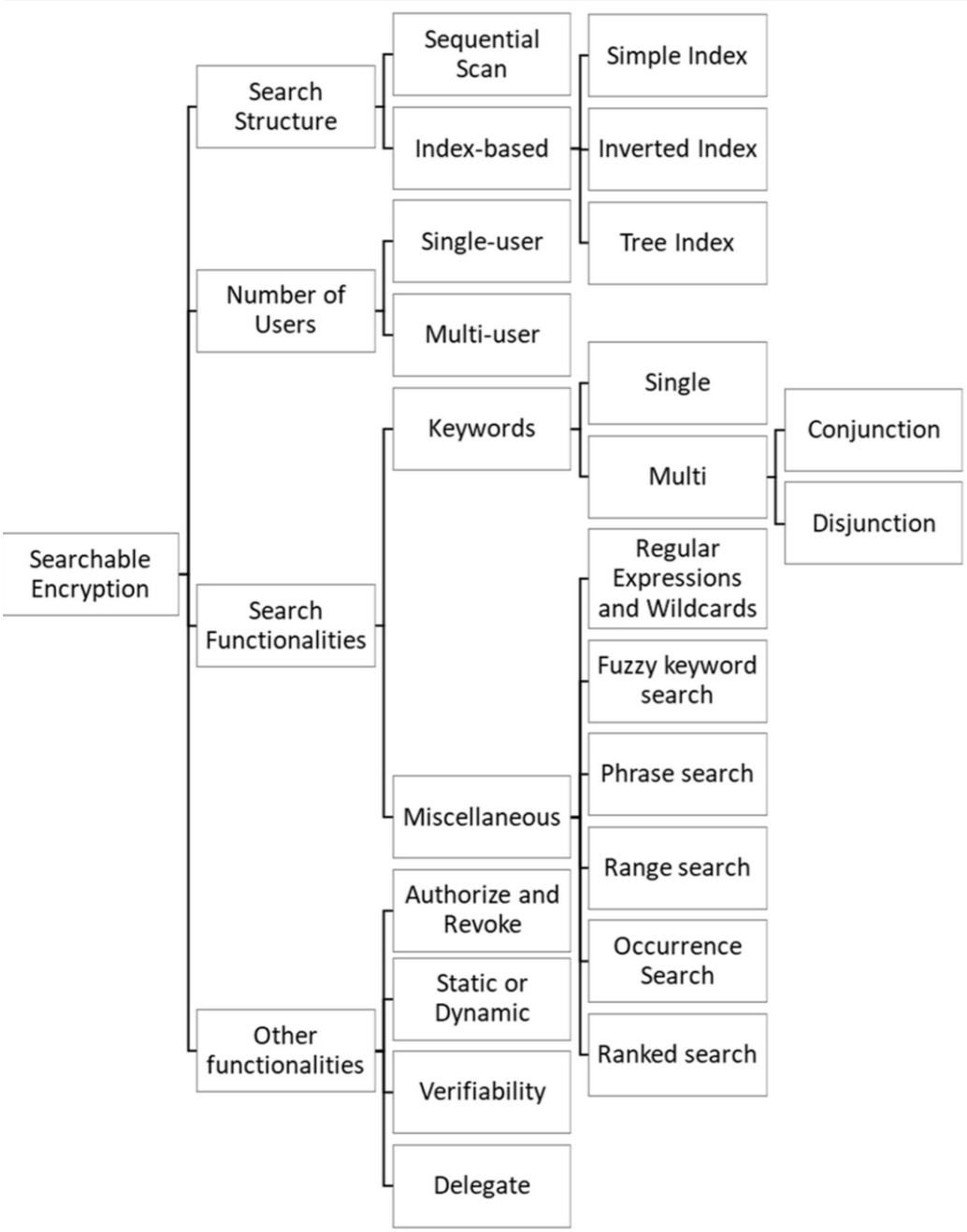

Figure 5.4: Key characteristics of a cloud-based SE scheme [80]

```
Alice - - - - - - - - - - - Bob - - - - - - - -      Time diff Location diff
E(TIME1)a E(Location1)a -> E(TIME1)b E(Location1)b  +100    +5
E(TIME2)a E(Location2)a -> E(TIME2)b E(Location2)b  +20     +3
E(TIME3)a E(Location2)a -> E(TIME3)b E(Location2)b  -1      +1
```

Figure 5.5: Google/Apple ID tracing

5.5 Location tracing

The EPIC contact tracing system uses homomorphic encryption in matching users up for possible contacts in a defined time window [81]. First, Alice defines data in time stamps and stores homomorphically encrypted timestamps for her location:

```
E(TIME1)a E(Location1)a
E(TIME2)a E(Location2)a
E(TIME3)a E(Location2)a
```

where E(TIMEx)a is the homomorphically encrypted timestamp value, and E(Locationx)a is the homomorphically encrypted location information. Then, Alice and Bob upload their homomorphically encrypted time stamp and location information to the Health Autority (HA), who stores these values:

```
E(TIME1)a E(Location1)a
E(TIME2)a E(Location2)a
E(TIME3)a E(Location2)a
E(TIME1)b E(Location1)b
E(TIME2)b E(Location2)b
E(TIME3)b E(Location2)b
```

The HA cannot tell either the time stamp or the location information. Alice is now identified as having COVID-19, and the server can identify her encrypted values and runs a homomorphic difference on the timestamps and location (Figure 5.5).

Here the HA cannot tell where Bob and Alice were and at what time, but they can tell that there was a match for a 1 s difference and if they were 1 m away from each other. In this way, Bob could be informed of a possible infection. Other information is also stored, which can be used for the matching process, such as the device type, the SSID of the wireless

Timestamp	u_1 $rssi$	m	Type	u_2 $rssi$
1509240563.03	-64	D8:84:66:4C:D1:00	WiFi	-85
1509240563.03	-69	D8:84:66:4E:E4:F0	WiFi	-79
1509240563.03	-59	D8:84:66:4E:F0:04	BL	-91

Figure 5.6: Google/Apple ID tracing

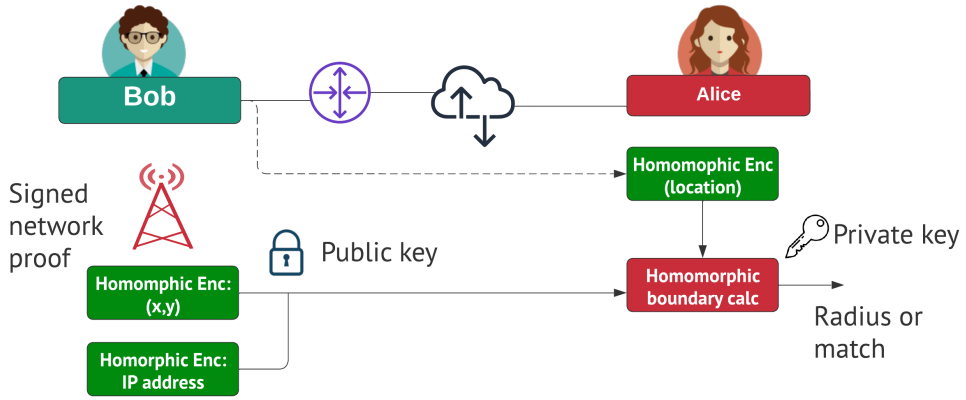

Figure 5.7: Privacy-aware location tracking

access point that they connected to, and the RSSI (Received Signal Strength Indication), as shown in Figure 5.6.

In TraceSecure [82], the authors proposed two private contact tracing methods using Bluetooth signals. The first one extends the TraceTogether application [83] by integrating a secure message-based protocol, while the second one incorporates a public key infrastructure that elaborates additive homomorphic encryption.

Within the tracking of an object, we can encrypt data from base stations and then determine the distance between the object and the base station, and thus find the nearest base station. This is illustrated in Figure 5.7.

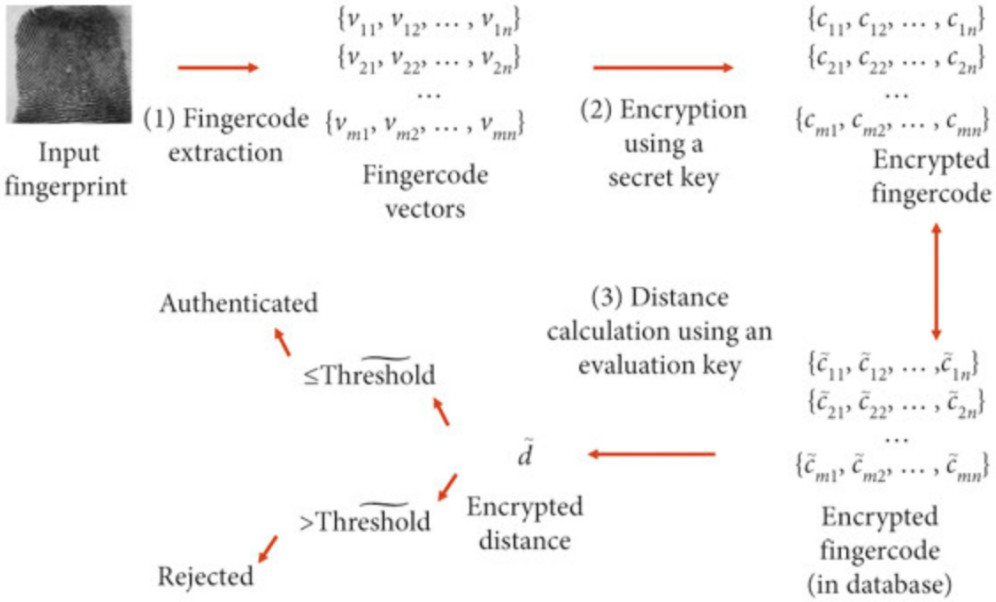

Figure 5.8: Privacy-aware location tracking

5.6 Privacy-aware biometrics

One application of a cloud-based system with homomorphic encryption is with biometric (and ID) matching [84]. With this, we create a vector for a biometric and then match it to the closest vector (Figure 5.8). For this, we compute a distance vector between the target and the vector stored in a database.

5.7 Apple Secure Cloud

Apple has developed a privacy-aware Cloud infrastructure which integrates machine learning. For this they use a Brakerski-Fan-Vercauteren (BFV) HE scheme for operations such as dot products and cosine similarity on embedding vectors for ML implementations. that are common to ML workflows. This supports 128-bit security within a classical and quantum setting.

Two applications include private information retrieval (PIR) for matches to data and for ML when looking for approximate matches with private nearest neighbor search (PNNS).

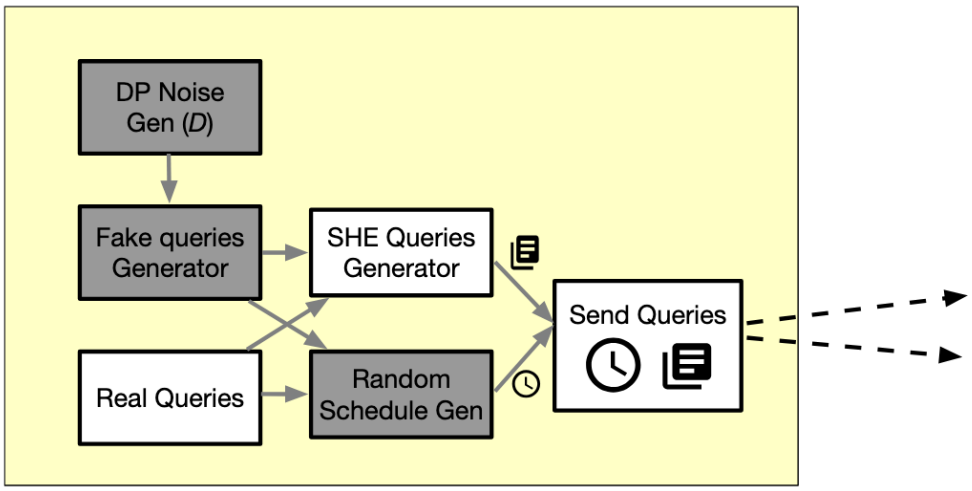

Figure 5.9: Differential privacy-related changes to hide real queries [85]

5.7.1 Private Information Retrieval (PIR)

With private information retrieval (PIR), we have a keyword-value database lookup. The client then aims to provide a keyword, and where the value is then returned. With client then encrypts the keyword and sends this to the server. The server then performs a homomorphic computation between the ciphertext and its database keyword values. Once matched, the server returns the encrypted value associated with the keyword. The client can then decrypt this. In this way the server never learns the keyword which has been used nor the retrieved result. An example of a search could be URL, and where Apple does not know about the URL which is being searched for.

5.7.2 Private Nearest Neighbor Search (PNNS)

With PNNS, Apple is able to find the nearest approximate match to a vector. This system is known as Wally [85], and can process eight million queries in less than two hours. With this, the client encrypts a vector and sends this to the server. The server then finds the nearest vector based on the nearest neighhour. In order to preserve privacy, we implement differential privacy, and where Wally receives fake calls, and where the system cannot determine if it is dealing with real or fake calls (Figure 5.9).

Figure 5.10 outlines an example of Apple's usage of holomorphic encryption. In this a user searchers for images within a given location. Apple

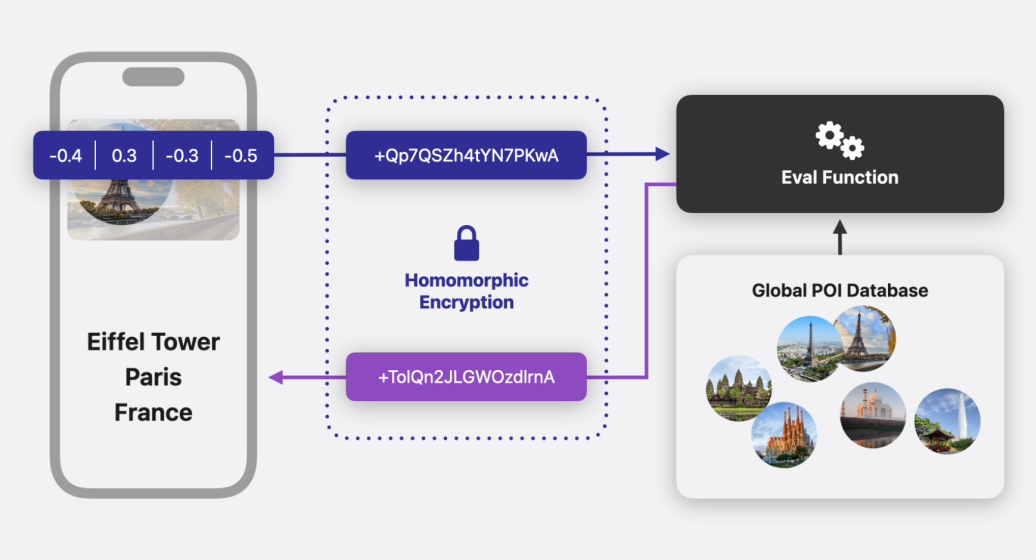

Figure 5.10: Using Private Nearest Neighbor Search for Enhanced Visual Search for photos [86]

then has a database of popular landmarks, and where their GPS location is encrypted with homomorphic encryption. The on-phone machine learning then searches for a landmark in a photo. If it finds one, it then encrypts a vector for the landmark and will add a region of interest. This is passed to the server, and which finds the closest encrypted vector to the target. This image is then returned to the client. In order that the server does not determine the image which is requested, differential privacy is used to create fake queries for the client, so that the real one cannot be matched to the client. This is done through an OHTTP (Oblivious HTTP) relay, and which also hides the IP address of the client.

Chapter 6

Secure multi-party computation (MPC)

6.1 Introduction

Secure Multi-Party Computation (MPC) enables two or more parties to collaboratively perform computations on their private data, obtaining the final result without revealing any party's sensitive input. Since its inception in the 1980s, beginning with Yao's pioneering work on two-party garbled circuits [87], MPC techniques have evolved from a theoretical curiosity into a foundational tool for building large-scale privacy-preserving applications. Over the decades, these techniques have seen substantial advancements, significantly reducing computational and communication costs. MPC's capabilities hold enormous potential for enabling a range of privacy-preserving applications, including machine learning applications that remain impractical due to stringent data privacy regulations.

Indeed, HE plays a major role in several advanced MPC protocols, particularly in scenarios where parties wish to minimize interaction or offload computations to untrusted third parties.

1. **Threshold Homomorphic Encryption in MPC:** Many MPC protocols use threshold homomorphic encryption schemes, where the decryption key is distributed among multiple parties. This approach allows parties to jointly perform computations on encrypted data and collectively decrypt the result, without any single party having access to the entire decryption key. For instance, the SPDZ protocol (pronounced speedz) [88] leverages additive homomorphic encryption to generate preprocessed data for use in MPC computations.

2. **Fully Homomorphic Encryption (FHE) for Outsourced MPC:** FHE allows for arbitrary computations on encrypted data. In the context of MPC, this enables secure outsourcing of computations to untrusted third parties. For example, the TFHE [62] scheme has been used to construct efficient MPC protocols where parties can delegate complex computations to a cloud server without revealing their inputs.

3. **Privacy-Preserving Machine Learning:** In scenarios where multiple parties want to train a machine learning model on their combined data without revealing individual datasets, HE-based MPC protocols play a crucial role.

6.2 Key Concepts and Terminology

6.2.1 Secret Sharing

In simple terms, a (t, n)-secret sharing scheme divides a secret s into n shares, ensuring that no information about s is revealed with fewer than t shares, while any collection of t or more shares can fully reconstruct the secret [89].

In Multi-Party Computation (MPC), secret sharing is a core building block for securely distributing inputs among participants. By splitting each party's private input into shares and distributing them to all other parties, the computation can be carried out on the shares without revealing the original inputs. The key advantage of secret sharing in MPC is that it allows the parties to collaborate on computing a function without any single party knowing the complete input data. This guarantees input privacy and makes secret sharing particularly well-suited for secure distributed computations. When combined with HE, secret sharing ensures that even partial data in transit remains protected, providing an extra layer of security.

For example: Suppose we have a secret $s = 123$, and we want to divide this secret into 5 shares (i.e., $n = 5$) such that any 3 of them (i.e., $t = 3$) are sufficient to reconstruct the secret.

Step 1: Polynomial Generation

The secret is represented as the constant term of a polynomial of degree $t-1$. For $t = 3$, we use a quadratic polynomial:

$$P(x) = s + a_1 \cdot x + a_2 \cdot x^2$$

where s is the secret, and a_1 and a_2 are randomly chosen coefficients.

Let $a_1 = 45$ and $a_2 = 67$, so the polynomial becomes:

$$P(x) = 123 + 45x + 67x^2$$

Step 2: Generating Shares

To create the shares, we evaluate the polynomial at distinct non-zero points. Let's compute the shares at $x = 1, 2, 3, 4, 5$:

$$\begin{aligned}
P(1) &= 123 + 45(1) + 67(1)^2 = 235, \\
P(2) &= 123 + 45(2) + 67(2)^2 = 481, \\
P(3) &= 123 + 45(3) + 67(3)^2 = 861, \\
P(4) &= 123 + 45(4) + 67(4)^2 = 1375, \\
P(5) &= 123 + 45(5) + 67(5)^2 = 2023.
\end{aligned}$$

Thus, the shares are:

$$(1, 235), (2, 481), (3, 861), (4, 1375), (5, 2023)$$

Step 3: Reconstructing the Secret

To reconstruct the secret, we need any 3 shares. Let's use $(1, 235)$, $(2, 481)$, and $(3, 861)$.

Using Lagrange interpolation, we reconstruct the polynomial $P(x) = c_0 + c_1x + c_2x^2$. The constant term c_0 gives us the secret:

$$P(0) = s = 123$$

Thus, the secret $s = 123$ is successfully reconstructed.

6.2.2 Threshold Schemes

A threshold scheme is a cryptographic protocol where a function or task can only be performed when a threshold number t of participants collaborate. The concept encompasses secret sharing but applies to a broader range of cryptographic operations, such as signing, decryption, or computation [90]. In these schemes, no single entity holds complete control over the operation, ensuring that a coalition of participants is required.

For instance, in threshold encryption, decryption can only be performed if at least t participants provide their shares of the decryption key. Similarly,

in threshold signature schemes, a valid digital signature is produced only if t signers cooperate.

Threshold encryption is a powerful application of threshold schemes, particularly when combined with homomorphic encryption (HE). HE allows computations to be performed directly on encrypted data without revealing the underlying plaintext. By combining threshold encryption and HE, we can create systems where multiple parties can collaboratively decrypt encrypted data without any single party gaining access to the plaintext.

Consider the following example of an MPC scheme implemented with homomorphic encryption: three parties—Bob, Alice, and Carol—generate their respective keys, and encrypted data is shared among them. Using `OpenFHE`, each participant computes their partial decryption. For instance, after encrypting data using Carol's public key, Bob, Alice, and Carol perform partial decryption using their secret keys. As shown in the code below, partial decryptions are computed for each participant, and these are then fused to recover the full plaintext:

```
// Perform partial decryptions by each party

auto ciphertextBob = cc->MultipartyDecryptLead(
    {ctAdd123}, bob.secretKey);

auto ciphertextAlice = cc->MultipartyDecryptMain(
    {ctAdd123}, alice.secretKey);

auto ciphertextCarol = cc->MultipartyDecryptMain(
    {ctAdd123}, carol.secretKey);

// Combine partial decryptions to recover plaintext

std::vector<Ciphertext<DCRTPoly>> partialCiphertextVec = {
    ciphertextBob[0], ciphertextAlice[0], ciphertextCarol[0]};

cc->MultipartyDecryptFusion(
    partialCiphertextVec, &plaintextMultipartyNew);
```

In this scenario, *Bob, Alice, and Carol's combined partial decryptions* enable the full decryption of the ciphertext, demonstrating the threshold nature of the scheme. This convergence of HE and MPC not only enhances

security by distributing trust among multiple participants but also maintains data privacy during computations, making it a powerful tool for secure and collaborative cryptographic tasks.

6.2.3 Verifiable Secret Sharing (VSS)

Verifiable Secret Sharing (VSS) is a cryptographic technique that enhances traditional secret sharing schemes by enabling participants to verify that their shares of a secret are valid and correctly generated [91]. In a standard secret sharing protocol, a secret (like a cryptographic key or sensitive data) is divided into multiple shares distributed among participants, where a subset of those shares is needed to reconstruct the original secret. VSS adds an additional layer of security by allowing participants to confirm the integrity and authenticity of the shares they receive. VSS plays a crucial role in Multi-Party Computation (MPC), where multiple parties collaborate to compute a function while keeping their inputs private. VSS ensures that all participants can trust the shares they receive and can participate confidently in the computation. By integrating HE, verifiable computations can be performed directly on encrypted shares, ensuring end-to-end data confidentiality throughout the entire process.

6.3 Practical Threshold encryption

With MPC (Multiparty Computation) we can split a task into a number of secure computations. In this case, we will use threshold encryption to split the decryption process into three encryption key shares. Overall, we will compute the addition of three data sets using homomorphic encryption and then allow Bob, Alice and Carol to generate decrypted values, which can be aggregated together to provide the result. This is illustrated in Figure 6.1 and implemented in OpenFHE [92].

6.3.1 Coding examples

The following are coding examples using OpenFHE:

- BFV with Threshold Encryption using OpenFHE and C++. Coding: Here.
- CKKS with Threshold Encryption using OpenFHE and C++. Coding: Here.

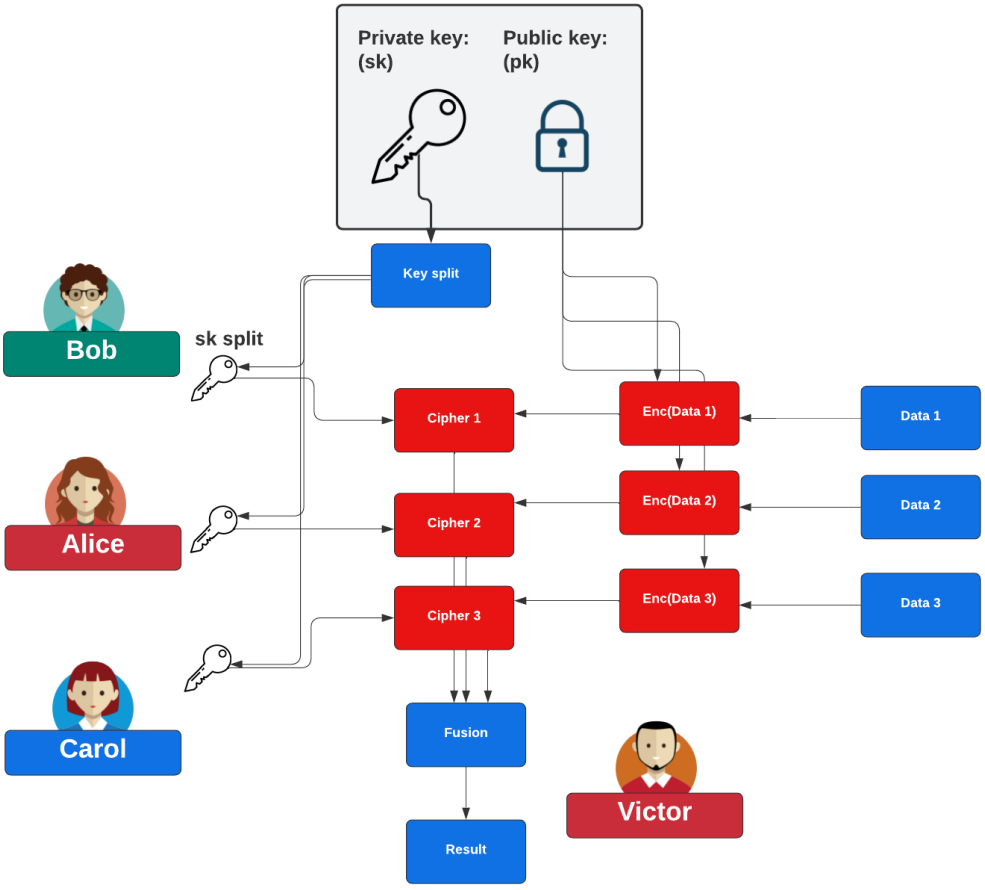

Figure 6.1: Threshold encryption

- Shamir Shares with BFV for Threshold Encryption using OpenFHE and C++. Shamir Shares with BFV for Threshold Encryption using OpenFHE and C++. Coding: Here.
- Shamir Shares with CKKS for Threshold Encryption using OpenFHE and C++. Shamir Shares with CKKS for Threshold Encryption using OpenFHE and C++. Coding: Here.

Chapter 7

Machine Learning with Homomorphic Encryption

7.1 Introduction

Homomorphic encryption supports the usage of machine learning methods, and some core features include a dot product operation with an encrypted vector and logistic functions. With this, OpenFHE supports a range of relevant methods and even has a demonstrator for a machine learning method.

7.2 State-of-the-art

Iezzi et al. [93] define two methods of training with homomorphic encryption:

- Private Prediction as a Service (PPaaS). This is where the prediction is outsourced to a service provider who has a pre-trained model and where encrypted data is sent to the service provider. In this case, the data owner does not learn the model used.
- Private Training as a Service (PTaaS). This is where the data owner provides data to a service provider and who will train the model. The service provider can then provide a prediction for encrypted data.

Wood et al [56] adds models of:

- Private outsourced computation. This involves moving computation into the cloud.
- Private prediction. This involves homomorphic data processed into the cloud, and not having access to the training model.

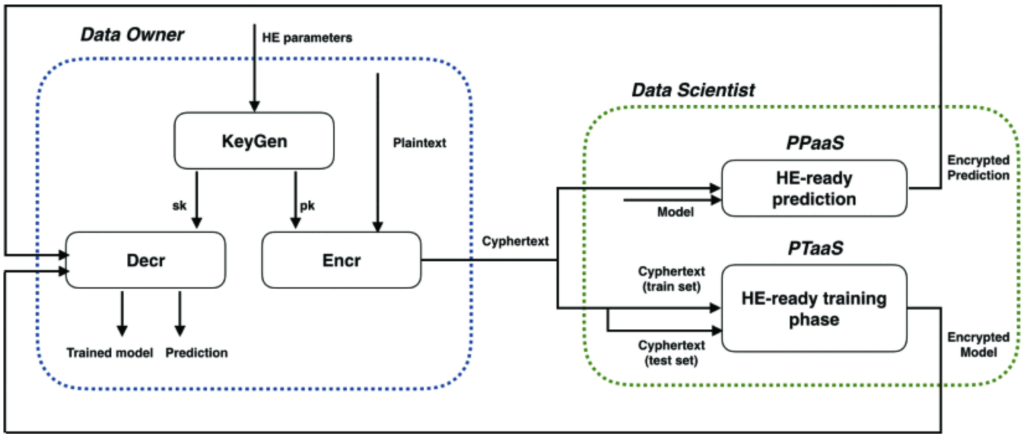

Figure 7.1: Machine Learning with HE [93]

- Private training. This is where a cloud entity trains a model based on the client's data.

The methods defined for Private Prediction as a Service solution are defined in Figure 7.2, and for Private Training as a Service solution in Figure 7.3.

7.3 Basic primitives

7.3.1 Logistic Function

With homomorphic encryption, we can represent a mathematical operation in the form of a homomorphic equation. One of the most widely used methods is to use Chebyshev polynomials, and which allows the mapping of the function to a Chebyshev approximation. A core application of the logistic function - also known as the sigmoid function - is within machine learning. With this, an artificial neural network is created with weighted summation and a sigmoid function. Mathematically, this is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{7.1}$$

For example, if we have $x = 0.5$, then, we get the mapping showing in Figure 7.4, and, in Python, we get:

Reference	PPaaS Model	HE scheme	Platform	Running Time	Dataset
Graepel et al. [54]	Linear Means Classifier	FV / Bra on SEAL	Intel Core i7 @2.8 GHz with 8GB of RAM	6 sec	Wisconsin Breast Cancer Data
	Fisher's Linear Discriminant classifier			20 sec	
Gilad-Bachrac et al. [32]	Prediction by NN with 5 layers	HE scheme based on [56] on SEAL	Intel Xeon E5-1620 @ 3.5GHz with 16GB RAM	570 sec for a prediction	MNIST
Costantino et al. [45]	Bag-of-word	BGV on HElib	Intel Core i7-6700 @ 3.40GHz with GB RAM	19 min	Tweets
Hesamifard et al. [34]	Prediction by Convolutional Neural Network with 6 layers	LHE on HElib	Intel Xeon E5-2640 @2.4GHz with 16GB RAM	320 sec	MNIST
				11686 sec	CIFAR-10
Masters et al. [31]	Nesterov's Accelerate GD-based logistic regression	CKKS on HElib	Titan V	4500 speed up on mult	Banco Bradesco financial transactions
Brutzkus et al. [33]	Same NN as [32] for prediction	BFV on SEAL	Azure standard VM with 8 vCPUs 32GB of RAM	0.29 sec for a prediction	MNIST
	Linear model trained with features generated by AlexNet			0.16 sec for prediction	CalTech-101
Al Badawi et al. [37]	Fasttext NN trained on plaintext data	CKKS on GPU	NVIDIA DGX-1 multi-GPU with 8 V100 cards	0.66 sec	AGNews
Podschwadt et al. [44]	Embedding layer + RNN layer with 128 units	CKKS on HElib	AMD Ryzen 5 2600 @ 3.5GHz with 32GB RAM.	547.6 sec for a batch of 128 samples	IMDb

Figure 7.2: Private Prediction as a Service solution [93]

Task	Reference	HE scheme	Platform	Running Time	Dataset
PTaaS - Logistic Regression	Gentry et al. [39]	BGV with bootstrapping on HElib	Intel Xeon E5-2698 v3 @2.30GHz with 250GB RAM	More than 4 hours; one hour if multithreading is used	iDASH competition data 2018
	Kim et al. [41]	CKKS	Intel Xeon CPU E5-2620 @2.10 GHz	6 mins	iDASH competition data 2017
	Chen at al. [55]	FV on SEAL	Intel(R) Xeon(R) CPU E3-1280 @ 3.70GHz with 16GB RAM	3.2 hours to 0.4 hours for 1-bit GD	iDASH competition data 2017
	Bergamaschi et al. [40]	CKKS on HElib	Intel E5-2640 @2.5GHz, with 64 GB RAM	20 min	iDASH competition data 2018
	Han et al. [9]	CKKS	IBM POWER8 @ 4.0GHz with 256GB RAM	17 hours	Korean credit bureau
				2 hours	MNIST
PTaaS - Neural Network	Nandakumar et al. [42]	BGV on HElib	Intel Xeon E5-2698 v3 @2.30GHz with 250GB RAM	1.5 days for NN1 (954 nodes, 50 epochs) 40 min for NN2 (122 nodes, 50 epochs)	MNIST (mini-batch of 60 training samples)
	Al Badawi et al. [37]	CKKS on SEAL	104 CPU cores	11.1 days (fasttext NN)	YouTube spam collection
		CKKS on GPU	NVIDIA DGX-1 multi-GPU with 8 V100 cards	5.04 days(fasttext NN)	

Figure 7.3: Private Training as a Service solution [**?**]

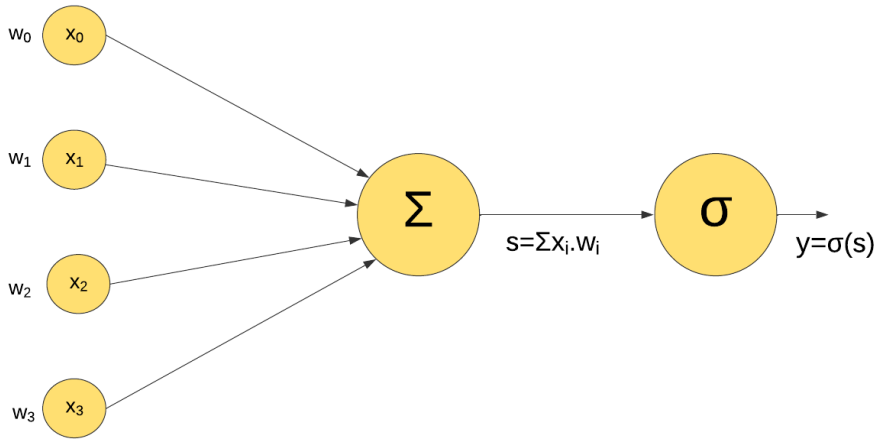

Figure 7.4: Sigmoid function

```
>>> import math
>>> x=0.5
>>> y=1/(1+math.exp(-x))
>>> print (y)
0.6224593312018546
>>> x=6
>>> print (y)
0.9975273768433653
```

This is supported in OpenFHE, and which implements Chebyshev approximation. For this, we can use the function of [94]:

```
Ciphertext EvalLogistic(ConstCiphertext ciphertext, double a,
                                double b, uint32_t degree)
                               const
```

and which evaluates 1/(1 + exp(-x)) for f(x), and where x is a range of coefficients with ciphertext. The value of a is the lower bound of the coefficients, and b is the upper bound. The degree value is the desired degree of approximation.

Logistic regression is often used to predict bi- nary outcomes of whether patients need treatment in medical applications, such as with diabetic patients [95].

7.3.2 Inner product

The inner product of two vectors of a and b is represented by $<a, b>$. It is the dot product of two vectors and represented as $<a, b> = |a|.|b|.cos(\theta)$, where θ is the angle between the two vectors. This operation is supported in OpenFHE [96].

7.3.3 Matrix operations

We can perform matrix operations with an encrypted input vector from OpenFHE [97]. For this, if we have a vector of the form:

$$v_1 = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \tag{7.2}$$

and a matrix of:

$$m_1 = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix} \tag{7.3}$$

We now get:

$$v_1.m_1 = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix} \tag{7.4}$$

and:

$$v_1.m_1 = \begin{bmatrix} x_1.w_{11} + x_2.w_{21} + x_3.w_{31} & x_1.w_{12} + x_2.w_{22} + x_3.w_{32} & x_1.w_{13} + x_2.w_{23} + x_3.w_{33} \end{bmatrix} \tag{7.5}$$

Thus we get:

$$y_1 = x_1.w_{11} + x_2.w_{21} + x_3.w_{31} y_2 = x_1.w_{12} + x_2.w_{22} + x_3.w_{32} y_3 = x_1.w_{13} + x_2.w_{23} + x_3.w_{33} \tag{7.6}$$

Figure 7.5 shows this setup.

7.4 Neural Network

Figure 7.5 is a 2-layer feed-forward network. We can now encrypt the input values with homomorphic encryption and use a plaintext version of our weightings (as these do not have to be secret). If we have a key pair of pk, sk, we can encrypt the input data vector (v_1) with:

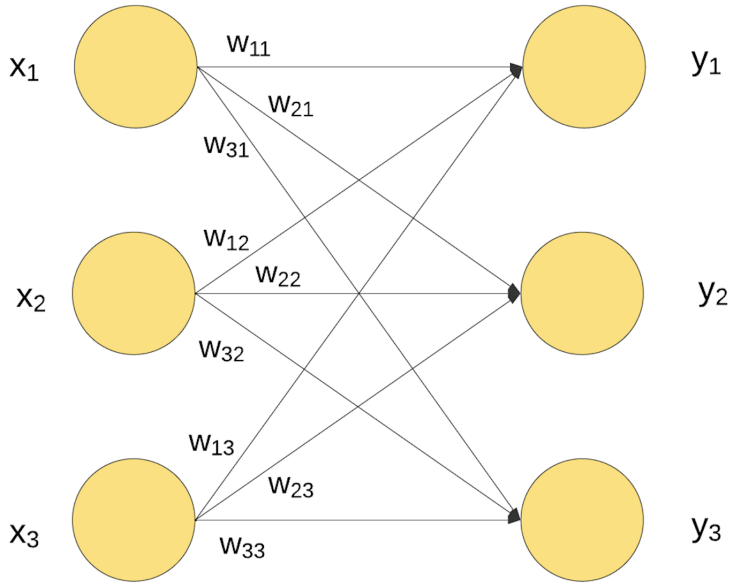

Figure 7.5: Neural network

$$v_1.m_1 = \begin{bmatrix} Ek_{pk}(x_1) & Ek_{pk}(x_2) & Ek_{pk}(x_3) \end{bmatrix} \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix} \tag{7.7}$$

We can then use the private key to determine:

$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = Ek_{sk}^{-1}(v_1.m_1) \tag{7.8}$$

We thus encrypt with the public key (pk) and decrypt the output with the private key (sk), as illustrated in Figure 7.6.

7.5 GWAS

Blatt et al. [98] implemented the Genome-wide association study (GWAS) and which is a secure large-scale genome-wide association study using homomorphic encryption.

7.5.1 Chi-Square GWAS

The Chi-squared GWAS test has been implemented in OpenFHE Here. With this, each of the participants in the student is given a public key from a

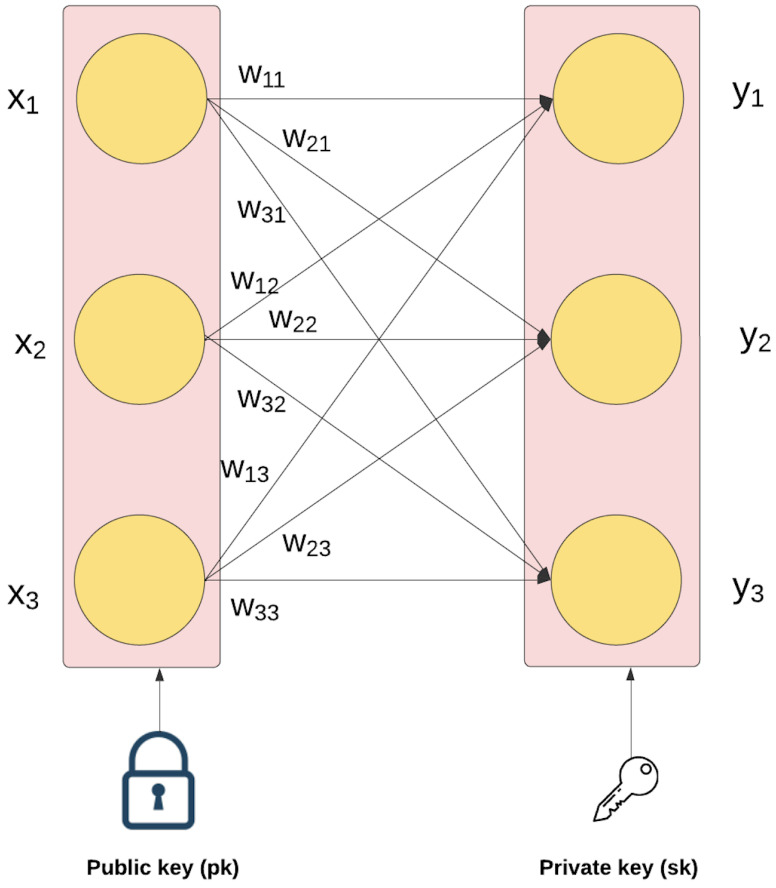

Figure 7.6: Neural network

GWAS (Genome-wide association studies) coordinator, and then encrypts the data with CKKS send sends it back for it to be processed (Figure 7.7). The computation includes association statistics using full logistic regression on each variant with sex, age, and age squared as covariates. Pearson's chi-square test uses categories and determines if there is a significant difference between sets of data. It implements as (Ref to the RunChi2 method from Here:

$$\tilde{\chi}^2 = \frac{1}{d} \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k} \tag{7.9}$$

and where:

- $\tilde{\chi}^2$ is the chi-square test statistic.
- O is the observed frequency.
- E is the expected frequency.

Overall the implementation involved a dataset of 25,000 individuals, and it was shown that 100,000 individuals and 500,000 single-nucleotide polymorphisms (SNPs) could be evaluated in 5.6 hours on a single server.

7.5.2 Linear Regression

The GWAS method is also implemented with linear regression for homomomorphic encryption (See RunLogReg in Here). The results (Figure 7.7) show that there was a good accuracy for both the Chi-squared and linear regression tests. The run time varied linearly with the number of participants in the test (Figure 7.9).

7.6 Support Vector Machines (SVM)

With the SVM (Support Vector Machine) model, we have a supervised learning technique. Overall, it is used to create two categories (binary) or more (multi) and will try to allocate each of the training values into one or more categories. Basically, we have points in a multidimensional space and try to create a clear gap between the categories. New values are then placed within one of the two categories.

Overall, we split out input data into training and test data and then train with an sklearn model with unencrypted values from the training data. The output from the model is the weight and intercepts. Next, we can encrypt

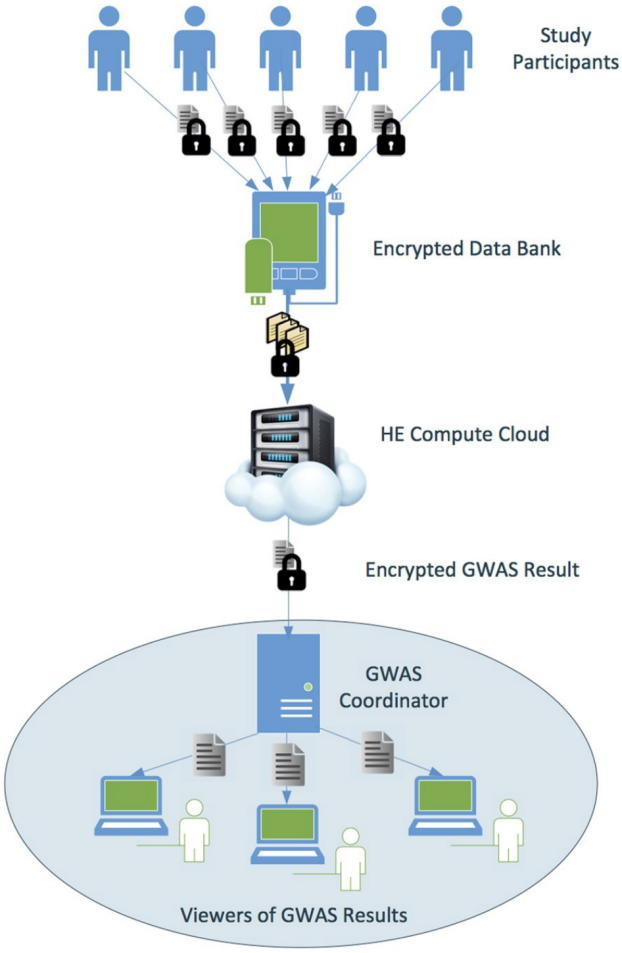

Figure 7.7: Chi-Square GWAS [98]

	GLM		HE LRA		HE Chisq	
SNP	OR	stat	OR	stat	OR	stat
rs10033900_T	1.09	1.97	1.08	1.91	1.06	1.44
rs943080_C	0.88	−2.94	0.89	−2.88	0.91	−2.26
rs79037040_G	0.88	−2.98	0.88	−2.91	0.89	−2.82
rs2043085_T	0.91	−2.01	0.92	−1.95	0.92	−2.13
rs2230199_C	1.41	6.83	1.38	6.67	1.40	7.10
rs8135665_T	1.12	2.04	1.12	2.03	1.12	2.29
rs114203272_T	0.62	−3.55	0.63	−3.50	0.67	−3.08
rs114212178_T	0.87	−0.70	0.87	−0.69	0.86	−0.77

Figure 7.8: Results for GWAS [98]

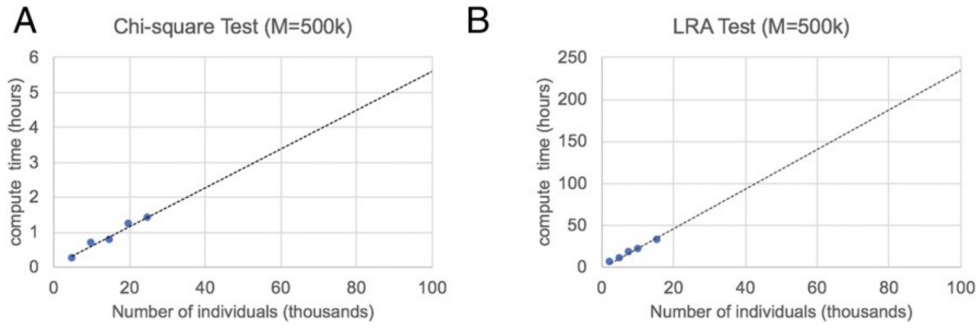

Figure 7.9: Results for GWAS [98]

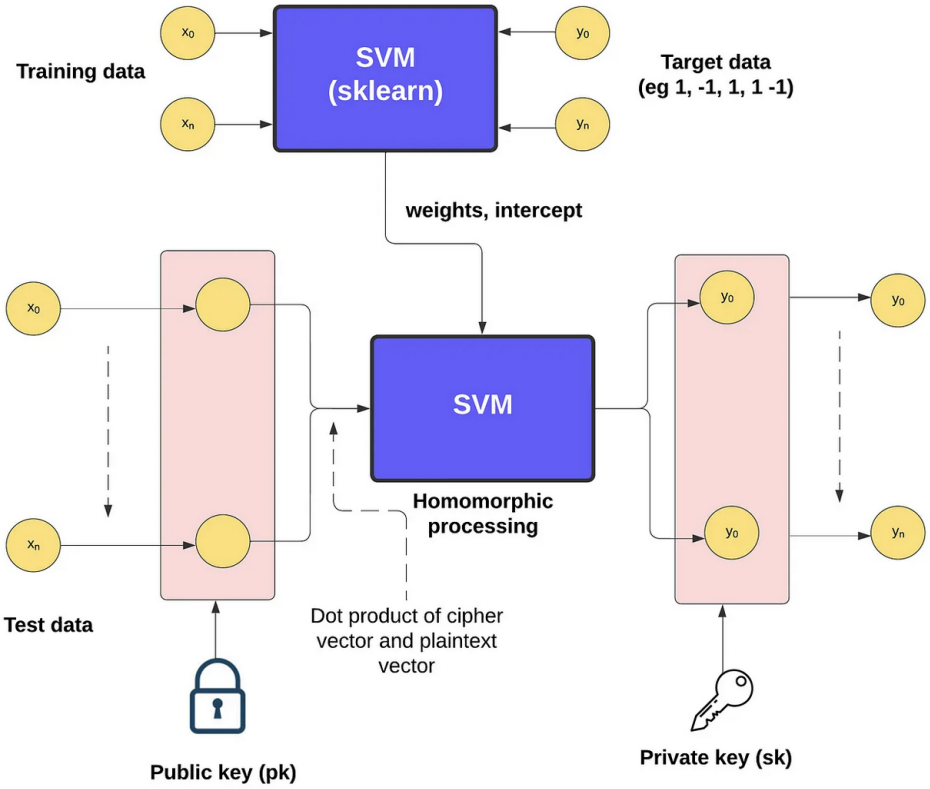

Figure 7.10: SVM

the test data with the homomorphic public key and then feed this into the SVM model. The output values can then be decrypted by the associated private key, as illustrated in Figure 7.10.

Overall, this involves a dot product of a cipher vector and a plaintext vector. First, we can use sklearn to train the model Here:

```
import pandas as pd
import numpy as np
from sklearn.svm import SVC

# Load the data
X_train = pd.read_csv('data/credit_approval_train.csv')
X_test = pd.read_csv('data/credit_approval_test.csv')
y_train = pd.read_csv('data/credit_approval_target_train.csv'
                                    )
y_test = pd.read_csv('data/credit_approval_target_test.csv')
```

```
# Model Training
print("---- Starting Models Training ----")

print("Starting SVM Linear")
svc_linear = SVC(kernel='linear')
svc_linear.fit(X_train, y_train.values.ravel())
print("SVM Linear Completed")

svc_poly = SVC(kernel='poly',degree=3,gamma=2)
svc_poly.fit(X_train, y_train.values.ravel())
print("SVM Poly Completed")

print("---- Model Training Completed! ----")

decision_function = svc_linear.decision_function(X_test)
ytestscore = decision_function[0]

decision_function_poly = svc_poly.decision_function(X_test)
ytestscore_poly = decision_function_poly[0]

# Saving Results
np.savetxt("models/weights.txt", svc_linear.coef_)
np.savetxt("models/intercept.txt", svc_linear.intercept_)
np.savetxt("data/ytestscore.txt", [ytestscore])
np.savetxt("models/dual_coef.txt", svc_poly.dual_coef_)
np.savetxt("models/support_vectors.txt", svc_poly.
                                support_vectors_)
np.savetxt("models/intercept_poly.txt", svc_poly.intercept_)
np.savetxt("data/ytestscore_poly.txt", [ytestscore_poly])
```

This splits the input data into training and test data. The training data is then used to train the model with a linear and a polynomial SVM training model. It then outputs the model with a number of weights and intercept values. Next, we can run our homomorphic encryption method and take the training data (x), the weights, and the bias for processing [here]:

```
pt_x = cc.MakeCKKSPackedPlaintext(x)
pt_weights = cc.MakeCKKSPackedPlaintext(weights.tolist())
pt_bias = cc.MakeCKKSPackedPlaintext([intercept])
```

These values remain as plaintext values. We can then encrypt the training data with the public key [here]:

```
ct_x = cc.Encrypt(keys.publicKey, pt_x)
```

We then create an inner product with the cipher training data and the weights [here]:

```
ct_res = cc.EvalInnerProduct(ct_x, pt_weights,n)
```

An example of implementing a dot product is here. The output is then the multiplication (inner product) of the cipher values of the training data and the weights. Next, we can mask out the first value with [here]:

```
mask = [0] * n
mask[0] = 1
pt_mask = cc.MakeCKKSPackedPlaintext(mask)
ct_res = cc.EvalMult(ct_res, pt_mask)
```

Then we add the bias [here]:

```
ct_res = cc.EvalAdd(ct_res, pt_bias)
```

Finally, we can decrypt the resultant value with the private key:

```
result = cc.Decrypt(ct_res, keys.secretKey)
```

7.7 Deep Neural Network

Wood et al. [56] defines a range of methods for Private (Deep) Neural Network Evaluation, as described in Figure 7.11.

7.7.1 Coding examples

A practical coding example is:

- BFV Inner Product using OpenFHE and C++. BFV Inner Product using OpenFHE and C++. The inner product of two vectors of a and b is represented by ⟨a,b⟩. Coding: Here.
- BFV Inner Product for Vectors of Varing Size using OpenFHE and C++. BFV Inner Product for Vectors of Varing Size using OpenFHE and C++. Coding: Here.
- Matrix Multiplication with Homomorphic Encryption for BFV using OpenFHE and C++. Matrix Multiplication with Holomorphic Encryption for BFV using OpenFHE and C++. I It uses an inner product method. Coding: Here.
- Vector/Matrix/Matrix Multiplication with Homomorphic Encryption for BFV using OpenFHE and C++. Vector/Matrix/Matrix Multiplication with Homomorphic Encryption for BFV using OpenFHE and C++. Coding: Here.

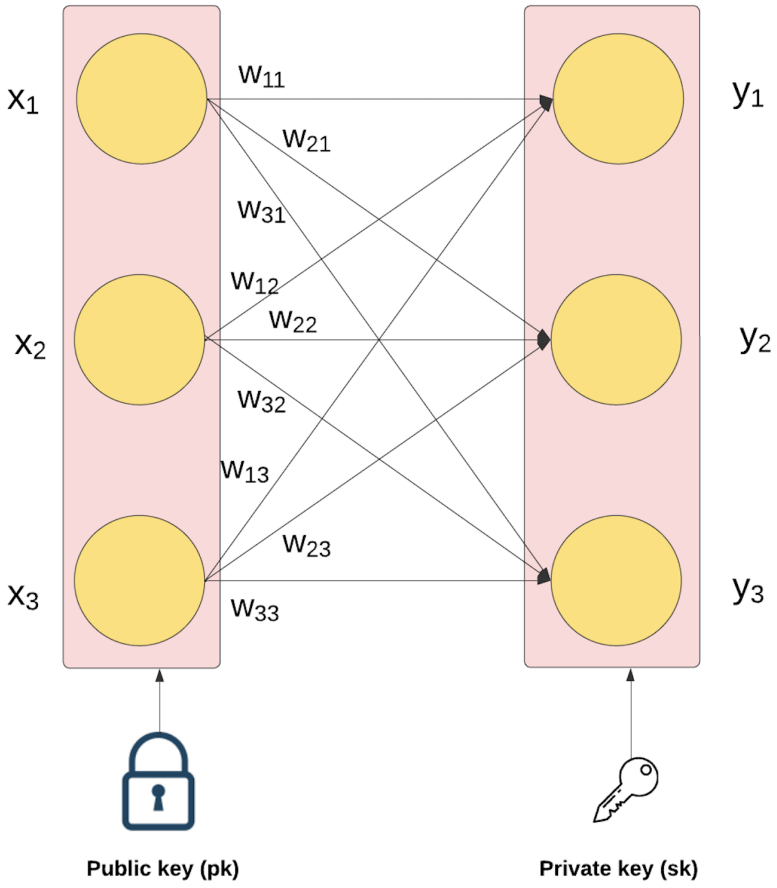

Figure 7.11: Frameworks for Private (Deep) Neural Network Evaluation Using FHE

- Matrix Multiplication with Homomorphic Encryption for CKKS using OpenFHE and C++. Matrix Multiplication with Homomorphic Encryption for CKKS using OpenFHE and C++. Coding: Here.
- Vector/Matrix/Matrix Multiplication with Homomorphic Encryption for CKKS using OpenFHE and C++. Vector/Matrix/Matrix Multiplication with Homomorphic Encryption for CKKS using OpenFHE and C++. Coding: Here.

7.8 Concrete ML

An alternative to OpenFHE is Concrete ML Here. It is an open-source Python-based implementation for FHE. The core library integration is concrete.ml.sklearn. A number of practical examples is here. It is install with:

```
pip install concrete-ml
```

7.8.1 Logistic Regression — Sklearn Breast Cancer Data set

This uses a simple data set for breast cancer data, and

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets

data = datasets.load_breast_cancer()

bc = pd.DataFrame(data.data, columns = data.feature_names)
bc['class'] = data.target

from mlxtend.feature_selection import
                                SequentialFeatureSelector as
                                SFS
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression

cv = KFold(n_splits=10, random_state=None, shuffle=False)
classifier_pipeline = make_pipeline(LogisticRegression())

sfs1 = SFS(classifier_pipeline,
           k_features=20,
           forward=True,
```

```
            scoring='neg_log_loss',
            cv=cv
            )

sfs1.fit(bc.drop(columns='class'), bc['class'])
sfs1.subsets_
```

We then integrate with concrete.ml.sklearn to implement linear regression here:

```
import numpy as np
from sklearn.model_selection import train_test_split
from concrete.ml.sklearn import LogisticRegression as
                                ConcreteLogisticRegression
from sklearn.preprocessing import RobustScaler

# Below are the ten features that contribute the most to
                                model accuracy

selected_features_10 = [
    'mean compactness',
    'mean concave points',
    'radius error',
    'area error',
    'worst texture',
    'worst perimeter',
    'worst area',
    'worst smoothness',
    'worst concave points',
    'worst symmetry'
]

logreg = LogisticRegression()
logreg.fit(X_train, y_train)

y_pred_test = np.asarray(logreg.predict(X_test))
sklearn_acc = np.sum(y_pred_test == y_test) / len(y_test) *
                                100

q_logreg = ConcreteLogisticRegression(n_bits={"inputs": 5, "
                                weights": 2})
q_logreg.fit(X_train, y_train)

q_logreg.compile(X_train)

q_y_pred_test = q_logreg.predict(X_test)
quantized_accuracy = (q_y_pred_test == y_test).mean() * 100

q_y_pred_fhe = q_logreg.predict(X_test, execute_in_fhe=True)
```

```
homomorphic_accuracy = (q_y_pred_fhe == y_test).mean() * 100

print(f"Regular Sklearn model accuracy: {sklearn_acc:.4f}%")
print(f"Clear quantised model accuracy: {quantized_accuracy:.
                                    4f}%")
print(f"Homomorphic model accuracy: {homomorphic_accuracy:.4f
                                    }")
```

7.8.2 Other machine learning models

Comcrete-ML uses a direct integration of concrete.ml.pandas for encrypting and decrypting a Pandas DataFrame here:

```
from concrete.ml.pandas import ClientEngine
from io import StringIO
import pandas

data_left = """index,total_bill,tip,sex,smoker
1,12.54,2.5,Male,No
2,11.17,1.5,Female,No
3,20.29,2.75,Female,No
"""

# Load your pandas DataFrame
df = pandas.read_csv(StringIO(data_left))

# Obtain client object
client = ClientEngine(keys_path="my_keys")

# Encrypt the DataFrame
df_encrypted = client.encrypt_from_pandas(df)

# Decrypt the DataFrame to produce a pandas DataFrame
df_decrypted = client.decrypt_to_pandas(df_encrypted)
```

Concrete-ML also allows for a direct replacement for the following methods:

- Linear models: LinearRegression, LogisticRegression, LinearSVC, LinearSVR PoissonRegressor, TweedieRegressor, GammaRegressor, Lasso, Ridge, ElasticNet and SGDRegressor.
- Tree Models: DecisionTreeClassifier, DecisionTreeRegressor, RandomForestClassifier, and RandomForestRegressor.
- NeuralNetClassifier (MLPClassifier) and NeuralNetRegressor (MLPRegressor).

- Nearest Neighbour. KNeighborsClassifier

An example of using SGDClassifier is here:
‘

```
from concrete.ml.sklearn import SGDClassifier
parameters_range = (-1.0, 1.0)

model = SGDClassifier(
    random_state=42,
    max_iter=50,
    fit_encrypted=True,
    parameters_range=parameters_range,
)
```

Chapter 8

Quantum resilience

8.1 Introduction

Our existing public key methods have been shown to have weaknesses in the advent of quantum computers. This includes risks to public key encryption from Shor's quantum factoring algorithm [99] and for symmetric key methods with Grover's method [100].

8.2 NIST PQC Competition

To address the weaknesses of current public key encryption methods, at the end of 2022, NIST selected a range of methods to replace existing key exchange and digital signature methods. The selected methods were CRYSTALS-Kyber for a key encapsulation method (GroKEM) and public key encryption (PKE), and CRYSTALS-Dilithium, Falcon and SPHINCS+ for digital signatures. The key exchange methods reached Round 3 for the assessment process and have now progressed to Round 4, with BIKE, Classic McEliece and HQC being assessed as alternatives to CRYSTALS-Kyber. For digital signatures, we reached Round 1, and where a number of additional signatures are being assessed. These can be classified as Multivariate Signatures, MPC-in-the-Head Signatures, Lattice-based Signatures, Code-based Signatures, Symmetric-based Signatures, Isogeny Signatures, and Other Signatures.

In the RSA method [2], the verification key consists of a public exponent e and modulus N where N is a product of two secret prime numbers p and q. The security of RSA relies on the difficulty of computing the factors of N. In 1994, Shor [101] proposed an efficient algorithm capable of finding the prime factors of any composite integer [102, 103, 104].

8.3 Learning With Errors

The security of LWE relies on two well-studied methods in lattice-based mathematics: Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). If we have a large number of dimensions, these problems are computationally intractable - even using quantum computers.

Chapter 9

Hardware requirements

9.1 Introduction

As with machine learning, homomorphic encryption uses vector and matrix operations. Along with this, we use modulo operations which can be larger than the registers used in typical processors. To overcome this, we can use devices which are more fully optimized to deal with large numbers and vector operations. These can include GPU processors, the Intel AVX-512 SIMD (Single Instruction Multiple Data) extensions, and specialist hardware.

9.2 Number Theoretic Transform (NTT)

Within homomorphic encryption, Number Theoretic Transform (NTT) can account for over 90% of the FHE computation time [105]. With NTT, we have inputs of $x_0, x_1 ... x_{n-1}$ and outputs of $y_0, y_1 ... y_{n-1}$, and where the values range from 0 to $m - 1$. α_n is the n-th primitive root of unity:

$$y_k = \sum_{j=0}^{n-1} x_j \alpha^{jk} \mod m \tag{9.1}$$

Jung et al [106] outlines that the parameters of HEAAN HE for Arithmetic of Approximate Numbers, and also known as CKKS [52]:

- L is the level, and is the number of multiplications that can be applied before we loose data.
- p is the rescaling factor.
- Q is the maximum ciphertext modulus.

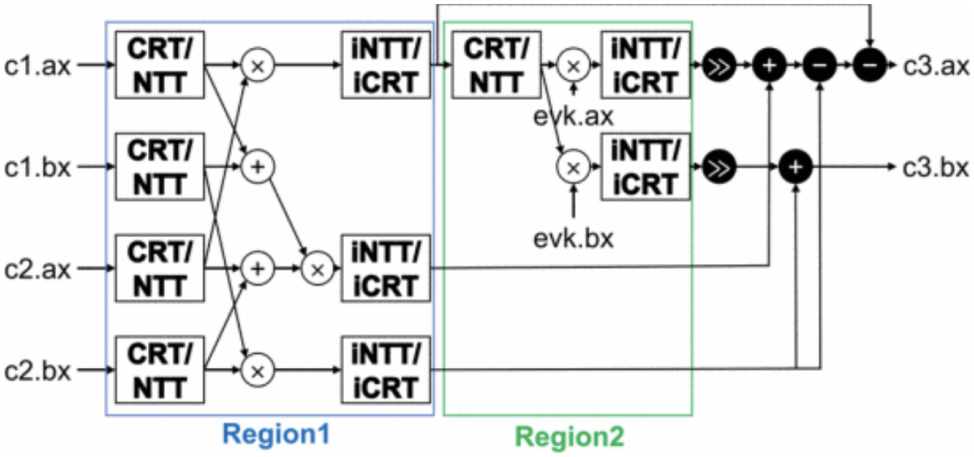

Figure 9.1: The overall flow of HE Multiplication in HEAAN [106]

- N is a power-of-two integer.

Overall, we have a ciphertext which has a modulus of Q and which is two polynomials of c.ax and c.bx. These are in a ring of $x^N + 1$. We can then perform a multiplication and then a rescaling process. For ciphertexts of c1 and c2, it is relatively simple of add the coefficients of the polynomials. With multiplication we compute a tensor product (Figure 9.1:

$$c3.ax = c1.axc2.ax, c3.cx = c1.bxc2.bx, c3.bx = c1.axc2.bx + c1.bxc2.ax \tag{9.2}$$

9.3 AVX-512

In 2013, Intel proposed 512-bit extensions for the Advanced Vector Extensions SIMD instruction set of the x86 architecture. It was released with the Intel Xeon Phi x200 (Knights Landing) processor, and now supported on a range of devices. AVX uses sixteen registers to perform a single instruction on multiple elements of data, including with 32-bit integers or four 64-bit floating-point values. Overall, AVX-512 is capable of dealing with 256-bit vector sizes.

Fu et al. [107] used AVX-512 to implement FHE and with a focus on the Number Theoretic Transform (NTT). For this, the research team implemented modular addition, subtraction, and multiplication for parallel opera-

```
// Parallel modular subtraction on 8 inputs, returning results in ch and cl
uint128_t submod128(__m512i* ch, __m512i* cl, __m512i ah, ..., __m512i ml) {
    t30 = _mm512_sub_epi64(al, bl);
    c1_m = _mm512_cmp_epu64_mask(al, bl, _MM_CMPINT_LT);
    t28 = _mm512_mask_add_epi64(bh, c1_m, bh, one);
    t29 = _mm512_sub_epi64(ah, t28);
    i28_m = _mm512_cmp_epu64_mask(ah, t28, _MM_CMPINT_LT);
    ...
    d3 = _mm512_add_epi64(d2, mh);
    *ch = _mm512_mask_blend_epi64(i28_m, t29, d3);
    *cl = _mm512_mask_blend_epi64(i28_m, t30, d1);
}
```

Figure 9.2: Modular subtraction C code using AVX-512 [107]

tions. This involves a scalar algorithm with 64-bit integers and then translate each arithmetic and bitwise operation to an AVX-512 equivalent. Figure 9.2 and Figure 9.3 outline the subtraction method with AVX-512. Test results shows that the vectorized code ran around 35 times faster than code on a GMP. When compared with the SPIRAL NTTX package [108], the code ran 2.2 times faster (Figure 9.4).

Jung et al [106] used AVX-512 (AVX-MT-24) to compare performance with a generalised GPU, and found an approximate 40 fold reduction in processing times.

9.4 GPU processing

To enhance the processing of homomorphic encryption, we can use GPUs. Ozcan et al. [109] performed an analysis using two GPUs: RTX306Ti (4,864 cores) and GTX 1080 (2,560 cores). Both have 8GB of RAM, and are compared with the Ryzen7 3800X CPU. The evaluation involved the assessment of add, multiply, re-linearisation and rotation for a range of n and $log_2(q)$ values (Figure 9.7). We can see there is a speed increase of around 10 to 50 times faster for the operations. In terms of power consumption, the GPU actually used less power for BFV multiplications (Figure 9.8).

The libraries used implement with Number Theoretic Transform (NTT) and inverse NTT and which optimize GPU kernel function calls. This integrations with the Microsoft SEAL library. For a ring dimension 2^{14} and a modulus bit size of 438, provides a 63.4 times speed-up comparing GPUs implementations over a CPU. The assessment of using NTT operations is

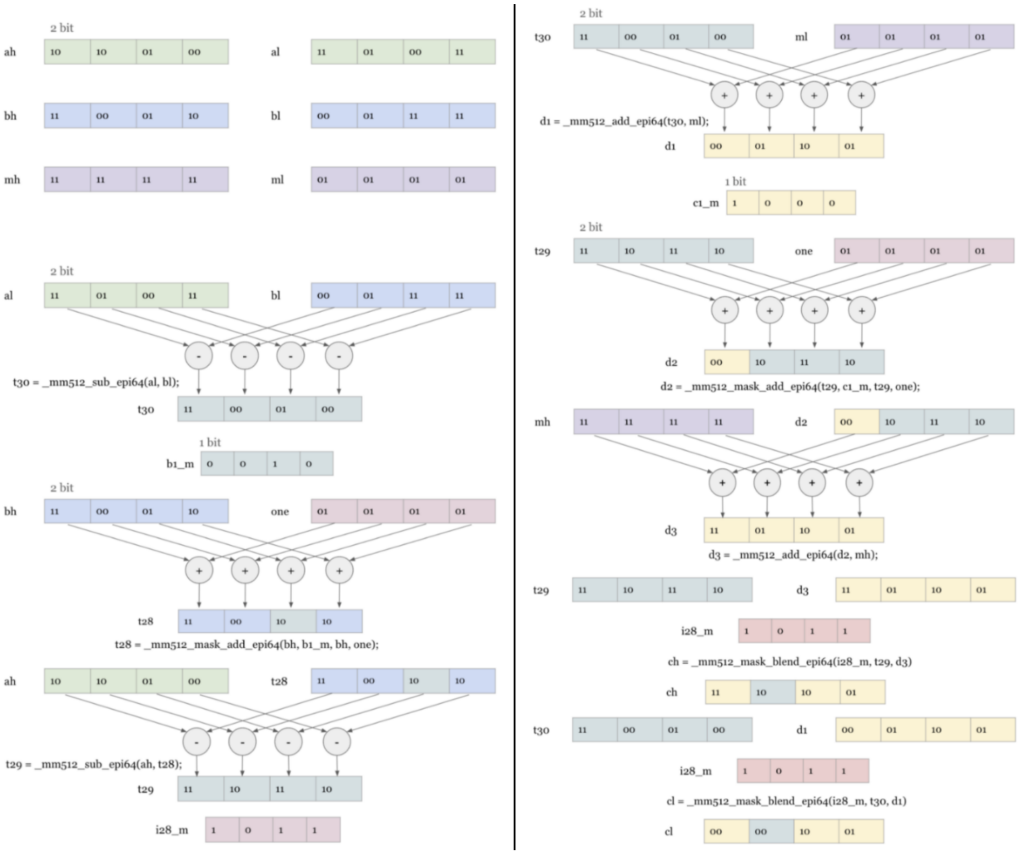

Figure 9.3: Overview of modular subtraction using AVX-512 [107]

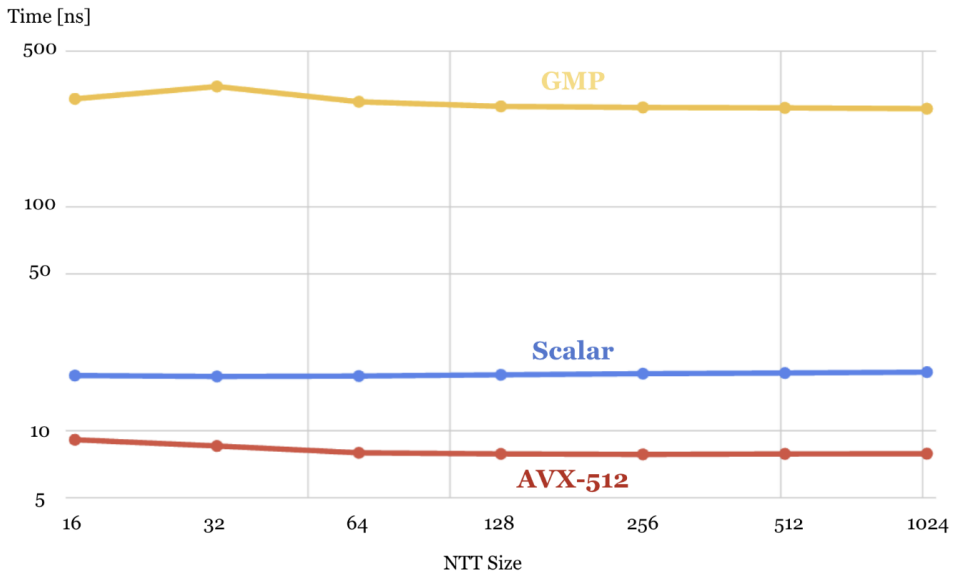

Figure 9.4: Runtime per butterfly for various NTT sizes [107]

	Execution time (ms) and [Relative speedup]		
Func	**Ref**-1	**Ref**-24	**AVX-MT**-24
CRT	628.6	43.5 [14.4×]	13.5 [46.6×]
NTT	576.5	27.6 [20.9×]	7.8 [73.6×]
iNTT	582.7	28.2 [20.7×]	8.7 [66.6×]
iCRT	1636.6	65.7 [24.9×]	44.1 [37.1×]
Extra	162.3	13.1 [12.4×]	13.1 [12.3×]
Total	3586.6	178.0 [20.1×]	87.3 [**41.1**×]

Figure 9.5: Comparing the execution time of HE Multiplication [106]

Feature	CPU	GPU	
		RTX3060Ti	GTX1080
Model	Ryzen7 3800X	RTX3060Ti	GTX1080
Threads	16	4864	2560
Freq.	4.20 GHz	1665 MHz	1733 MHz
RAM	32 GB (3600 MHz)	8 GB	8 GB
Mem. Type	-	GDDR6	GDDR5X
Mem. Bus	-	256 bits	256 bits
Bandwidth	-	448 GB/s	320 GB/s

Figure 9.6: CPU and GPUs [109]

given in Figure 9.9.

9.4.1 Costings

As we see from the previous section, we see a speed-up of between 10 and 50 times with a GPU over a CPU. As shown in Figure 9.10, the cost of a single GPU with one GPU is $1 per hour for 24 GB GPU memory and 16 GB RAM, while an eight GPU cluster costs around $16 per hour.

9.5 FHE-specific processors

While GPUs and SIMD extensions can speed up the process, some researchers outline methods that build architectures which will optimize the processing of homomorphic encryption. Figures 9.11 and 9.12 show the F1 architecture from Samardzic et al. [110], and Figure 9.13 shows the operation of vector multiplication. The team have since advanced their architecture on with Craterlake [111] (Figure 9.13).

In terms of chip design Soni et al. [112] outline the RPU architecture (Figure 9.15 and which is optimized for Ring Learning With Errors solutions. Overall they found that the RPU could fit on a $20.5mm^2$ chip and provided a 1,485 times speed-up over a CPU running a 64k, 128-bit NTT workload.

			GPU with [29] NTT		GPU with new NTT		[22]	SEAL	T
Operation	**n**	$\log_2 q$	RTX3060Ti	GTX1080	RTX3060Ti	GTX1080	Tesla V100	CPU	T_s
Add.	2^{12}	109	4 μs	4.6 μs	4 μs	4.6 μs	-	14 μs	3.5×
	2^{13}	218	5.1μs	6.2 μs	5.1 μs	6.1 μs	-	58 μs	11.37×
	2^{14}	438	**12.3 μs**	19.4 μs	**12.3 μs**	19.4 μs	-	**233 μs**	**18.94**×
	2^{15}	881	44 μs	64.2 μs	44 μs	64.2 μs	-	778 μs	17.68×
Mult.	2^{12}	109	172 μs	259 μs	86 μs	155.8 μs	-	3212 μs	37.3×
	2^{13}	218	297 μs	532 μs	202 μs	423.4 μs	-	11883 μs	58.8×
	2^{14}	438	**1037 μs**	2294 μs	**768 μs**	1856.1 μs	-	**48757 μs**	**63.4**×
	2^{15}	881	5372 μs	10657 μs	3757 μs	-	-	205295 μs	54.6×
Relin.	2^{12}	109	46 μs	82.7 μs	39.51 μs	59.3 μs	-	625 μs	15.81×
	2^{13}	218	104 μs	145 μs	88.54 μs	143.4 μs	-	3100 μs	35.01×
	2^{14}	438	**462 μs**	1013 μs	**376.61 μs**	825.3 μs	-	**18295 μs**	**48.57**×
	2^{15}	881	3530 μs	6651 μs	3150 μs	-	-	111736 μs	35.47×
Rot.	2^{12}	109	51 μs	87 μs	42.1 μs	59.4 μs	-	642 μs	15.24×
	2^{13}	218	116 μs	172 μs	103.3 μs	162.7 μs	-	3157 μs	30.56×
	2^{14}	438	**544 μs**	1339 μs	**458.7 μs**	1067.2 μs	-	**18338 μs**	**39.97**×
	2^{15}	881	3879 μs	10504 μs	3464.5 μs	-	-	113437 μs	32.74×
Mult. + Relin.	2^{12}	60	-	-	**136 μs**	-	**859 μs**	-	**6.31**×
	2^{13}	120	-	-	170 μs	-	1012 μs	-	5.95×
	2^{14}	360	-	-	661 μs	-	2010 μs	-	3.04×
	2^{15}	600	-	-	2875 μs	-	4826 μs	-	1.67×

Figure 9.7: Comparison results of SEAL BFV scheme operations and literature with our GPU implementations of BFV scheme operations [109]

		CPU		GPU	
n	**Count**	**Power***	**Time**	**Power***	**Time**
2^{12}	1	55.5 W	3025 $\boldsymbol{\mu s}$	44.37 W	92 μs
	10	60.49 W	30430 $\boldsymbol{\mu s}$	44.5 W	830 μs
	100	60.65 W	317682 μs	44.54 W	9890 μs
	500	63.55 W	1369910 μs	44.49 W	51246 μs
2^{13}	1	55.71 W	9121 μs	44.15 W	161 μs
	10	59.52 W	89266 $\boldsymbol{\mu s}$	44.25 W	1589 μs
	100	61.39 W	852098 μs	44.24 W	16275 μs
	500	64.78 W	3970960 μs	47.05 W	81714 μs
2^{14}	1	57.23 W	38414 μs	44.35 W	829 μs
	10	61.31 W	381921 $\boldsymbol{\mu s}$	44.37 W	7515 μs
	100	60.8 W	3763901μs	44.61 W	71858 μs
	500	64.16 W	18786647 μs	48.60 W	337236 μs
2^{15}	1	59.55 W	186334 μs	44.29 W	3382 μs
	10	59.46 W	1796976 $\boldsymbol{\mu s}$	45.48 W	36301 μs
	100	60.89 W	17822724 μs	55.71 W	338212 μs
	500	66.93 W	89749372 μs	65.13 W	342199 μs

Figure 9.8: Power consumption of BFV multiplication on CPU and GPU [109]

		32 Bit (Implemented On RTX 3060Ti)						64 Bit (Implemented On RTX 3060Ti)					
		Forward NTT			Inverse NTT			Forward NTT			Inverse NTT		
n	NTT_count	[29]	T.W.	T	[29]	T.W.	T	[29]	T.W.	T	[29]	T.W.	T
2^{12}	4	12.3 μs	**11 μs**	**1.12**×	11.2 μs	**11.1 μs**	**1.11**×	19.5 μs	**14.3 μs**	**1.36**×	16 μs	**15.4 μs**	**1.03**×
	16	13 μs	**11.2μs**	**1.16**×	12.2 μs	12.2 μs	1×	22.5 μs	**17.2 μs**	**1.30**×	17.8 μs	19.4 μs	0.91×
	32	13.9 μs	17.9 μs	0.77 ×	18.4 μs	19.2 μs	0.96×	23.5 μs	25.2 μs	0.93×	29.3 μs	**26.6 μs**	**1.10**×
	64	23.1 μs	28.6 μs	0.80×	29.5 μs	**29.3 μs**	**1**×	39.9 μs	43 μs	0.93×	52.9 μs	**50.1 μs**	**1.05**×
	128	38.9 μs	46.7 μs	0.83×	48.1 μs	48.7 μs	0.98×	75.7 μs	81.6 μs	0.92×	96.5 μs	**91.1 μs**	**1.06**×
2^{13}	4	17.1 μs	**12.2 μs**	**1.40**×	14.1 μs	14.3 μs	0.98×	24.4 μs	**16.6 μs**	**1.47**×	21.1 μs	**20.5 μs**	**1.03**×
	16	18.4 μs	18.4 μs	**1**×	22.2 μs	**20.3 μs**	**1.09**×	28.2 μs	**25.6 μs**	**1.10**×	34.8 μs	**28.6 μs**	**1.21**×
	32	28.6 μs	29.4 μs	0.97×	34.9 μs	**30.3 μs**	**1.15**×	47.9 μs	**44.4 μs**	**1.08**×	59.4 μs	**49.9 μs**	**1.19**×
	64	49.8 μs	**46.7 μs**	**1.07**×	57.3 μs	**50.7 μs**	**1.13**×	97.1 μs	**82.1 μs**	**1.18**×	121.2 μs	**93.9 μs**	**1.29**×
	128	88 μs	91.2 μs	0.96×	124 μs	**96.1 μs**	**1.29**×	170.7 μs	**156.4 μs**	**1.09**×	224.1 μs	**173.6 μs**	**1.29**×
2^{14}	4	20.1 μs	**15.2 μs**	**1.32**×	17.6 μs	**16.3 μs**	**1.08**×	29.98 μs	**21.76 μs**	**1.37**×	24.5 μs	**24.1 μs**	**1.01**×
	16	33.7 μs	**30.5 μs**	**1.05**×	40.9 μs	**30.1 μs**	**1.36**×	54.4 μs	**46.54 μs**	**1.17**×	65.9 μs	**53.2 μs**	**1.23**×
	32	57.3 μs	**50.1 μs**	**1.14**×	64.5 μs	**51.8 μs**	**1.24**×	121.8 μs	**84.6 μs**	**1.44**×	143.3 μs	**97.2 μs**	**1.47**×
	64	112.6 μs	**96 μs**	**1.17**×	147.4 μs	**99.3 μs**	**1.48**×	218.5 μs	**160.5 μs**	**1.36**×	266.8 μs	**180 μs**	**1.48**×
	128	210.7 μs	**176.1 μs**	**1.19**×	277.1 μs	**183 μs**	**1.51**×	420.8 μs	**303.19 μs**	**1.38**×	511.9 μs	**331.7 μs**	**1.54**×
2^{15}	4	25.6 μs	26.4 μs	0.96×	28.7 μs	**25.9 μs**	**1.10**×	35.8 μs	41.2 μs	0.86×	47.3 μs	50.3 μs	0.94×
	16	64.1 μs	**52.2 μs**	**1.22**×	74.7 μs	**53.2 μs**	**1.40**×	147.1 μs	**100 μs**	**1.47**×	170.1 μs	**95.6 μs**	**1.78**×
	32	136.3 μs	**100.3 μs**	**1.35**×	173 μs	**102.4 μs**	**1.69**×	266.2 μs	**191.8 μs**	**1.38**×	325.5 μs	**193.2 μs**	**1.68**×
	64	254.9 μs	**192.1 μs**	**1.32**×	322.2 μs	**193.6 μs**	**1.66**×	514.2 μs	**372.2 μs**	**1.38**×	633.7 μs	**377.7 μs**	**1.67**×
	128	491.1 μs	**362.4 μs**	**1.35**×	623.1 μs	**364.3 μs**	**1.71**×	998.9 μs	**709.3 μs**	**1.41**×	1202.9 μs	**725.2 μs**	**1.66**×

Figure 9.9: Timings of GPU implementation of NTT and inverse NTT operations [109]

	Instance Size	GPU	GPU Memory (GiB)	vCPUs	Memory (GiB)	Storage (GB)	Network Bandwidth (Gbps)	EBS Bandwidth (Gbps)	On Demand Price/hr*	1-yr ISP Effective Hourly (Linux)	3-yr ISP Effective Hourly (Linux)
Single GPU VMs	g5.xlarge	1	24	4	16	1x250	Up to 10	Up to 3.5	$1.006	$0.604	$0.402
	g5.2xlarge	1	24	8	32	1x450	Up to 10	Up to 3.5	$1.212	$0.727	$0.485
	g5.4xlarge	1	24	16	64	1x600	Up to 25	8	$1.624	$0.974	$0.650
	g5.8xlarge	1	24	32	128	1x900	25	16	$2.448	$1.469	$0.979
	g5.16xlarge	1	24	64	256	1x1900	25	16	$4.096	$2.458	$1.638
Multi GPU VMs	g5.12xlarge	4	96	48	192	1x3800	40	16	$5.672	$3.403	$2.269
	g5.24xlarge	4	96	96	384	1x3800	50	19	$8.144	$4.886	$3.258
	g5.48xlarge	8	192	192	768	2x3800	100	19	$16.288	$9.773	$6.515

Figure 9.10: AWS GPU Costings

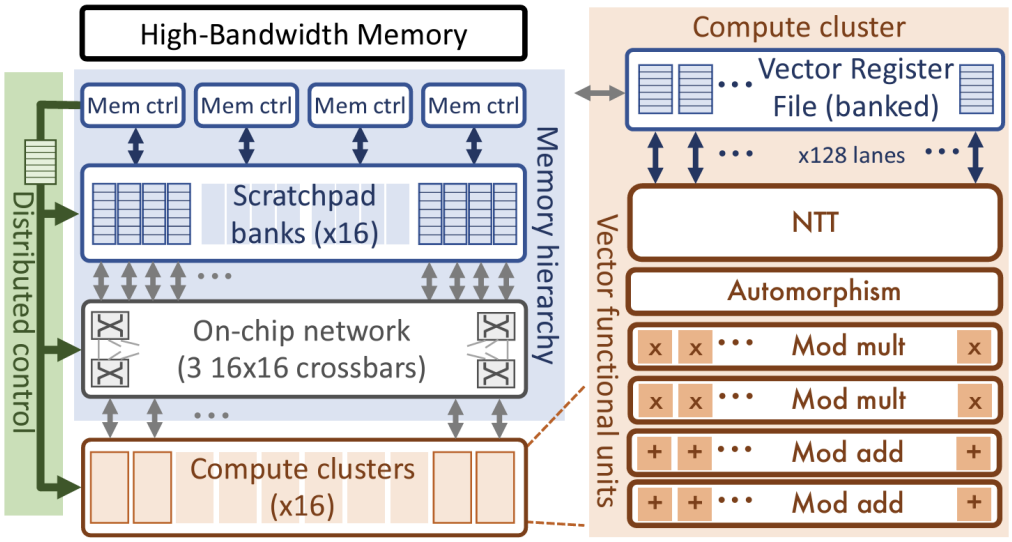

Figure 9.11: F1 architecture [110]

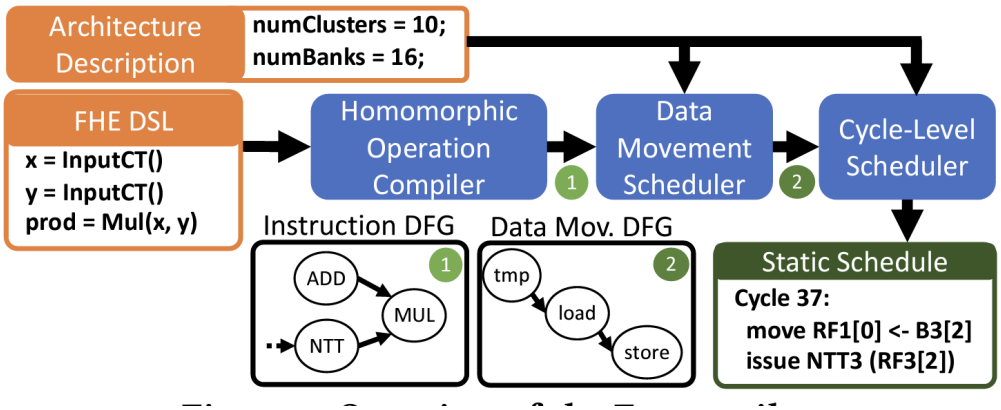

Figure 9.12: F1 vector operation [110]

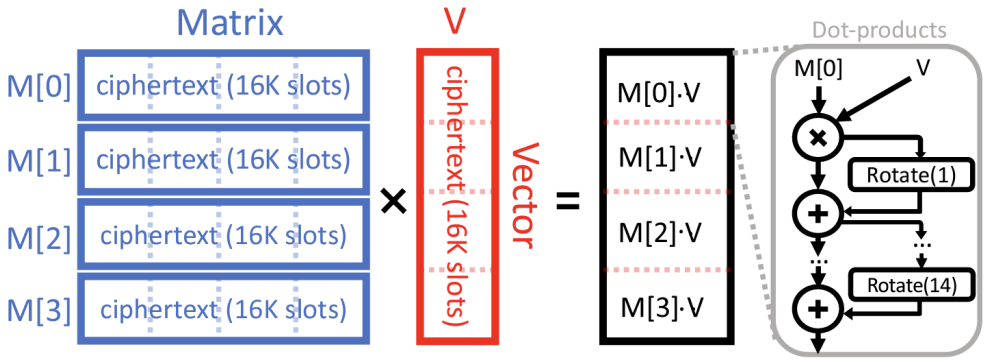

Figure 9.13: F1 Vector operation [110]

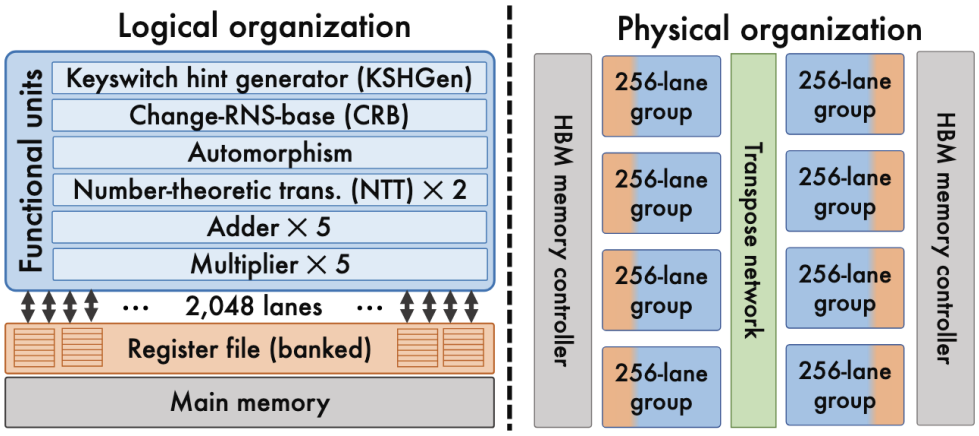

Figure 9.14: Craterlake architecture [111]

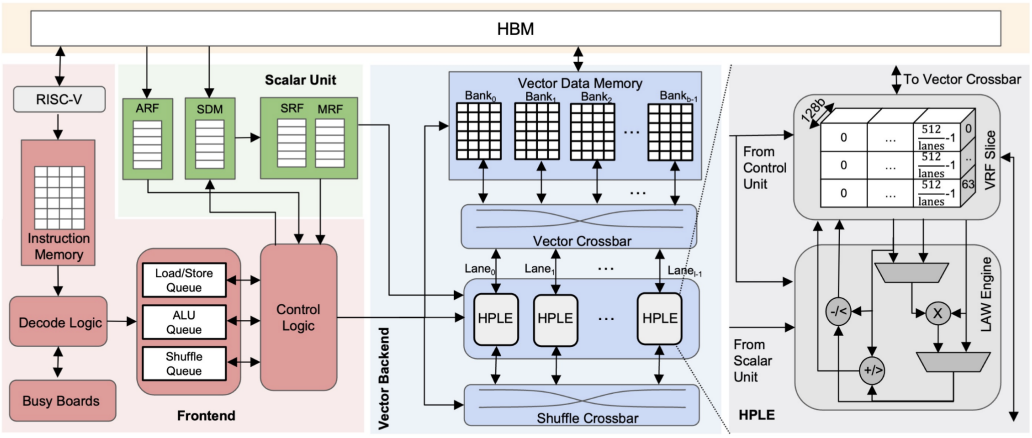

Figure 9.15: RPU architecture [112]

Chapter 10

Data fragmentation

10.1 Introduction

Within homomomorphic encryption, we can fragment data and allow for it to be transmitted, stored or processed within fragments.

10.2 Aggregation

With aggregation HE, we can split the data into a number of fragments, and then distribute these. Castelluccia et al [113] uses a parent and child architecture for an ad-hoc network. For this each node adds their routing hop data with homomorphic encryption. To encrypt a message(m) within a range of 0 to $M-1$, we generate a random keystream value of k (between 0 and $M-1$). To encrypt we compute:

$$c = Enc(m, k, M) = m + k(\text{mod } M) \tag{10.1}$$

and to decrypt:

$$Dec(c, k, M) = c - k(\text{mod } M) \tag{10.2}$$

For the addition of ciphertexts, we have:

$$c_1 = Enc(m_1, k_1, M) c_2 = Enc(m_2, k_2, M) \tag{10.3}$$

and simply add the ciphertext values:

$$c_1 + c_2 = Enc(m_1 + m_2, k_1 + k_2, M) \tag{10.4}$$

We thus have:

$$k = k1 + k2Dec(c1 + c2, k, M) = m1 + m2 \tag{10.5}$$

We can then compute a summation with:

$$C_x = \sum_{i=1}^{n} c_{x_i} (\text{mod } M) \tag{10.6}$$

The average can then be computed with:

$$S_x = Dec(C_x, K, M) = C_x - K (\text{mod } M) \tag{10.7}$$

and where:

$$K = \sum_{i=1}^{n} k_i \tag{10.8}$$

Kapusta et al [114] has since expanded the additive method to implement additively homomorphic encryption and fragmentation scheme (AHEF).

Chapter 11

Side Channel Analysis

11.1 Introduction

Side-channel analysis (SCA) refers to a category of attacks that exploit information leaked from the physical implementation of cryptographic algorithms, rather than weaknesses in the mathematical structures themselves [115] [116]. Unlike traditional cryptanalysis, which focuses on mathematical vulnerabilities, SCA leverages physical or observable characteristics of a system during operation - including timing variations, power consumption patterns, or electromagnetic emissions [117] [118] as depicted in Figure 11.1. These physical emanations can provide insights into sensitive data, such as cryptographic keys, and allow attackers to breach secure systems by observing these leaks rather than directly attacking the underlying algorithms. For example, during the encryption process in the RSA algorithm, the ciphertext is generated by:

$$c \equiv m^e \pmod{n} \tag{11.1}$$

where c, m, e, and n represent ciphertext, plaintext, key, and the product of p and q, respectively. To implement exponentiation in a real machine, the "Square and Multiply" algorithm is the easiest method. Algorithm 1 shows how "Square and Multiply" computes exponentiation. It scans the bits of the exponent e from left to right. If the current bit of e is one, there is an additional modular multiplication operation during the current loop. This additional operation leads to measurable computing power/time variation. Careful measurements and analysis can recover the secret key, bit by bit, as depicted in Figure 11.2.

Homomorphic encryption (HE), faces unique security challenges due to side-channel vulnerabilities which arise for several reasons including:

- **Computational Intensity:** HE operations are highly computation-

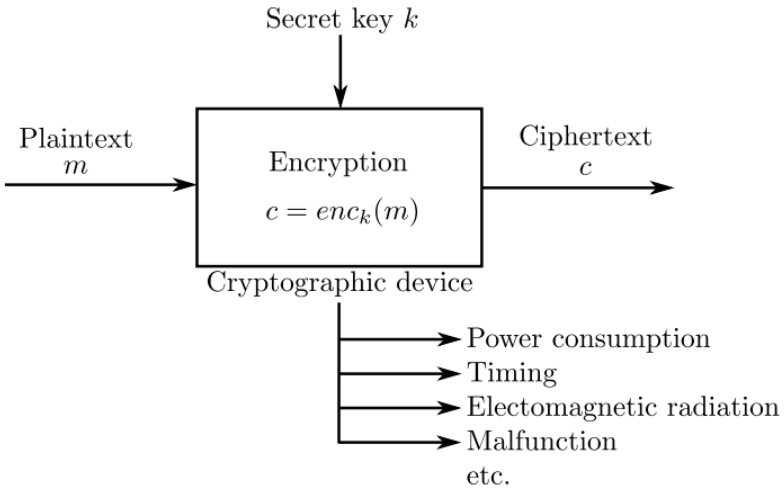

Figure 11.1: Possible side channels of a cryptographic device during an encryption [118].

Algorithm 1 Exponentiation by Square-and-Multiply

```
1: x ← 1
2: for each bit of e from left to right do
3:     x ← x^2 (mod n)
4:     if current bit of e is 1 then
5:         x ← x · m (mod n)
6:     end if
7: end for
8: return x
```

ally intensive, leading to longer exposure windows for side-channel analysis [120]. This extended processing time may offers attackers more opportunities to collect and analyze leaked information.

- **Complex Operations:** HE schemes often involve complex mathematical operations with execution patterns that may vary depending on the encrypted data or key material. These variations can result in distinguishable side-channel signatures [120].

- **Cloud Computing Context:** HE is frequently deployed in cloud environments for secure outsourced computations, introducing additional attack vectors. Malicious co-located processes can potentially exploit side-channel leakage through shared hardware resources[121].

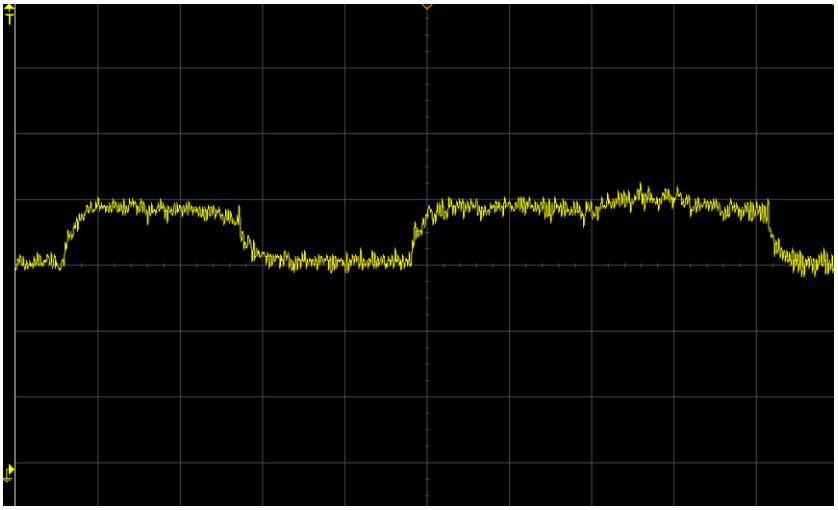

Figure 11.2: Observing RSA key bits using power analysis: The left peak shows the power consumption during the squaring-only step, the right (broader) peak shows the multiplication step, allowing exponent bits 0 and 1 to be distinguished [119].

A practical example of these challenges is the timing analysis of a homomorphically encrypted comparison operation. The time taken to compare encrypted values might vary based on the actual values, potentially leaking information about the encrypted data. This risk is especially concerning in privacy-sensitive applications, such as encrypted medical data analysis or secure financial computations.

11.2 Categories of Side-Channel Analysis

Side-channel analysis (SCA) is a powerful technique used by attackers to extract sensitive information from cryptographic implementations by observing physical characteristics like power consumption, timing information, electromagnetic radiation, or error responses. Eavesdroppers can monitor the power consumed during operations, the electromagnetic radiation emitted during decryption and signature generation, or the time taken to perform cryptographic operations. By analyzing these physical signals, attackers can infer secret information, such as cryptographic keys, and compromise the security of the system. Additionally, attackers can exploit how a cryptographic device behaves when errors occur, potentially revealing vulnerabilities in the implementation [117] [122].

Side-channel attacks can be broadly categorized into two main types:

passive side-channel attacks (also known as tamper attacks) and **active side-channel attacks**. Each type exploits physical information leakage in distinct ways [117]:

11.2.1 Passive Side-Channel Attacks

Passive side-channel attacks focus on observing physical signals emitted by a device during its normal operation without interfering with the device itself. Passive attacks are further divided into two subcategories [123]:

- **Simple Analysis Attacks:** These attacks rely on direct observations of leaked signals, such as power consumption or timing information. The attacker interprets these signals to deduce sensitive data, often requiring minimal computational effort.Popular examples of this are Simple Power Analysis (SPA), Simple Electromagnetic Analysis (SEMA), and Basic timing analysis
- **Differential Analysis Attacks:** These attacks involve statistical analysis of multiple measurements to extract subtle correlations between the physical characteristics and the secret data. Differential Power Analysis (DPA) is a well-known example of this type of attack, where the attacker correlates power traces with known plaintext or ciphertext to recover secret keys.

11.2.2 Active Side-Channel Attacks

Active side-channel attacks involve intentionally manipulating the target device to induce faults or abnormal behavior. By analyzing the device's response to such tampering, attackers can extract critical information, such as secret keys. These attacks exploit vulnerabilities in how cryptographic devices handle unexpected conditions.

Side-Channel Attacks could also be classified based on the type of information channel exploited as follows.

11.2.3 Power Analysis

Power analysis attacks exploit variations in power consumption during cryptographic operations and they could be categorized to the following key classes [124]:

- Simple Power Analysis (SPA): Simple Power Analysis (SPA) is a technique that involves directly analyzing power consumption measurements captured during encryption and decryption operations. By visually inspecting these power traces, attackers can often identify patterns that reveal information about the cryptographic algorithm's execution flow or even the secret key itself. This technique was pioneered by Paul Kocher and his colleagues, Jaffe and Jun, who also introduced Differential Power Analysis (DPA) in their influential paper .SPA can yield information about a devices operation as well as key material. SPA could be used to collect information about the targets cryptographic implementations by, e.g., interpret how many rounds are used during encryption/decryption. SPA is the simplest form of power analysis.

- Differential Power Analysis (DPA): Differential Power Analysis (DPA) is a sophisticated side-channel attack technique that involves statistically analyzing power consumption data collected during cryptographic operations. Unlike Simple Power Analysis (SPA), which relies on direct observation of power traces, DPA examines subtle correlations between power consumption and processed data over multiple cryptographic operations. In a DPA attack, the adversary uses hypothetical power models, such as the Hamming weight or Hamming distance, to predict power consumption patterns based on specific inputs or intermediate values. By comparing these predictions with actual power traces, the attacker can identify correlations and deduce sensitive information, such as cryptographic keys. DPA is particularly powerful because it can extract information even in the presence of noise and countermeasures, making it a significant threat to devices with inadequate side-channel protections.

- Correlation Power Analysis (CPA): Correlation Power Analysis (CPA) is an advanced side-channel attack technique that uses statistical correlation (e.g. Pearson correlation coefficient) to link power consumption patterns of a cryptographic device with hypothetical models, such as Hamming weight or Hamming distance. By analyzing multiple power traces recorded during cryptographic operations and comparing them to predicted power values for various key guesses, CPA identifies the key hypothesis with the highest correlation as the correct one. Its robustness to noise and precision make CPA a significant threat to cryptographic devices, necessitating countermeasures like randomization, noise injection, or constant power hardware designs to mitigate its effectiveness.

11.2.4 Timing Analysis

Timing analysis attacks exploit variations in the execution time of cryptographic operations to extract sensitive information [117]. By analyzing the relationship between timing fluctuations and the operations being performed, attackers can infer details about the cryptographic process. These attacks often target subtle discrepancies in hardware or software behavior. Common techniques include:

- Monitoring cache access patterns to infer data dependencies and memory usage.
- Analyzing branch prediction behavior to deduce control flow or conditional operations.
- Examining instruction scheduling delays to uncover computational bottlenecks or specific algorithmic steps.

11.2.5 Electromagnetic Analysis

Electromagnetic (EM) analysis leverages the electromagnetic emissions generated by a device during operation to extract or disrupt sensitive information [122]. This technique is versatile, with both passive observation and active interference methods.

11.2.6 Fault Analysis

Fault analysis, a sophisticated form of active attack, exploits errors intentionally introduced into cryptographic computations to extract sensitive information, such as secret keys or intermediate values [117]. Attackers manipulate the system's environment or operational conditions to disrupt its normal behavior. By analyzing the resulting faulty outputs, they can infer critical data through techniques like differential fault analysis or other statistical methods. Common fault injection methods include [122]:

- Voltage Glitching: Intentionally manipulating the power supply voltage to create transient voltage spikes or dips, causing the circuit to malfunction and potentially reveal sensitive information.
- Clock Manipulation: Altering the frequency or phase of the clock signal to disrupt the timing of operations, leading to incorrect calculations and potential security vulnerabilities.

- Laser/EM Injection: Employing focused laser beams or electromagnetic radiation to target specific circuit components and induce faults, such as bit flips or temporary circuit failures.

11.3 Case Studies: Side-channel Attacks on HE Systems

In the context of homomorphic encryption (HE), SCA poses unique challenges. While HE theoretically ensures robust cryptographic security, its practical implementations may leak exploitable side-channel information, particularly during computationally intensive operations like key generation, encryption, and homomorphic evaluation.

11.3.1 Single-Trace Attack on SEAL's BFV Encryption Scheme

The study in [120] introduced the first single-trace side-channel attack on homomorphic encryption (HE), specifically targeting the SEAL implementation of the Brakerski/Fan-Vercauteren (BFV) scheme prior to v3.6. The attack exploits power-based side-channel leakage during the Gaussian sampling phase of SEAL's encryption process, enabling plaintext recovery with a single power measurement. The proposed attack follows a four-step methodology to recover plaintext messages from SEAL's BFV encryption scheme. First, the attack identifies each coefficient index being sampled during the encryption process. Second, it extracts sign values from control-flow variations, which reveal key information about the sampled coefficients. Third, the attack recovers the coefficients with high probability using data-flow variations observed in the power trace. Finally, the Blockwise Korkine-Zolotarev (BKZ) algorithm is applied to explore and estimate the remaining search space, further refining the attack's success. The study also focused on recovering plaintext messages by extracting coefficients of error polynomials. These coefficients are integral to the encryption process, and their recovery undermines the cryptographic hardness of the scheme. Hence, the methodology identifies and exploits vulnerabilities in the `set_poly_coeffs_normal` function, responsible for error polynomial sampling. These vulnerabilities include branch operations that reveal sign information, and negation operations that reduce false positives by exploiting Hamming weight differences. Using real power measurements on a RISC-V FPGA implementation of SEAL v3.2, the attack reduces the security level of the plaintext encryption from 2^{128} to

$2^{4.4}$, highlighting the significant implications of side-channel attacks on HE. 'However, the attack has several drawbacks, including the need for profiling and key configuration, requiring many traces to create accurate templates [120]. It is also limited to a single device, and cross-device attacks may need machine learning for profiling. Furthermore, the attack was performed at a 1.5 MHz frequency, as higher frequencies increase noise and may require advanced equipment. Additionally, targeting more secure versions (196-bit or 256-bit keys) is harder due to increased precision and more coefficients.

11.3.2 Single-Trace ML Attack on CKKS SEAL's Key Generation

The authors in [125] uncover new side-channel vulnerabilities in Microsoft SEAL by focusing on its number theoretic transform (NTT) function using power analysis. Specifically, it presents an attack targeting the NTT operation within the SEAL CKKS scheme's key generation process. The study demonstrates that the NTT, used during key generation, leaks ternary values (-1, 0, $+1$) corresponding to secret key coefficients. The key innovation lies in developing a sophisticated two-stage neural network-based classifier capable of extracting side-channel information from a single measurement, demonstrating an unprecedented 98.6% accuracy in revealing secret key coefficients on the ARM Cortex-M4F processor. The research explores the impact of compiler optimizations, analyzing SEAL's NTT implementation across optimization levels from `-O0` (no optimization) to `-O3` (maximum optimization). While `-O3` eliminates previously identified vulnerabilities, the study reveals new side-channel leakages in the `guard` and `mul_root` operations under this setting. Random delay insertion, evaluated as a countermeasure, is shown to be ineffective against the proposed attack. This study distinguishes itself from prior side-channel analyses by targeting the latest version of SEAL (v4.1) and addressing vulnerabilities absent in other implementations. Unlike multi-trace attacks, which focus on decryption operations, this single-trace attack directly exploits NTT computations, bypassing defenses like masking. The study also reveals additional leakages with `-O3` optimization and refines attack accuracy. The findings underscore the need for robust countermeasures to secure FHE implementations like SEAL against single-trace side-channel attacks, particularly those targeting efficient and constant-time arithmetic.

11.3.3 Side-Channel Vulnerabilities in LWE/LWR-Based Cryptography

The authors in [126] demonstrated successful attacks against multiple post-quantum cryptography implementations, breaking through various side-channel countermeasures, including masking and shuffling techniques. Their work revealed that these vulnerabilities are not implementation-specific but rather stem from core algorithmic properties of LWE/LWR-based cryptography. While this research does not explicitly target homomorphic encryption schemes, its findings have significant implications for the broader family of LWE/LWR-based cryptographic systems, including homomorphic encryption. As many modern homomorphic encryption schemes are built upon these same mathematical foundations, they could potentially be vulnerable to similar side-channel attacks targeting specifically, the incremental storage of decrypted messages in memory and ciphertext malleability properties inherent to LWE/LWR-based schemes.

11.3.4 Cache-Timing Attack on the SEAL Homomorphic Encryption Library

The authors in [127] exposed a cache-timing vulnerability in the SEAL homomorphic encryption library, specifically in its implementation of Barrett modular multiplication. The researchers identified a timing side-channel in the non-constant-time implementation of extra-reductions using the ternary operator, which leaks information about the secret key. Leveraging a novel remote cache-timing methodology, the attack aims to recover the secret key involved in modular multiplications. By analyzing ciphertexts that cause an extra reduction, the researchers solve Diophantine equations to progressively narrow down the range of possible secret keys. The approach combines insights from Bézout's theorem with optimized enumeration techniques to refine key candidates efficiently. The attack requires as few as eight ciphertexts that trigger extra reductions to uniquely determine the secret key. To eliminate the vulnerability, the authors proposed a constant-time implementation of the `SEAL_COND_SELECT` macro. This replacement avoids conditional branching by using bitwise operations to compute results in constant time, ensuring that the timing of the operation no longer depends on the input values.

11.3.5 Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption

The authors in [128] introduced a single-trace side-channel attack targeting lattice-based cryptography, demonstrating its vulnerability to key recovery using observations from a single decryption. Unlike previous side-channel attacks, this approach is uniquely powerful because it can penetrate even masked implementations by recovering individual shares and subsequently reconstructing the complete decryption key. It focuses on the Number Theoretic Transform (NTT), a critical component in almost all efficient lattice-based cryptography implementations, making the attack applicable across a broad range of encryption schemes, including homomorphic encryption schemes. Unlike previous differential power analysis (DPA) attacks, which overlooked NTT, the authors leverages its less-protected nature. The attack comprises three main steps: (1) side-channel template matching on modular operations during the inverse NTT; (2) combining intermediate probabilities via belief propagation (BP) on the FFT-like NTT structure, optimized to make BP computationally feasible; and (3) utilizing leaked intermediate values along with the public key to recover the private key through lattice decoding.

11.3.6 A Practical Full Key Recovery Attack on TFHE and FHEW by Inducing Decryption Errors

The study in [129] introduced the latest side-channel attack targeting Fully Homomorphic Encryption (FHE) schemes on the server side. Unlike prior attacks that focused on the client side, this study demonstrates that a malicious server can inject carefully calculated perturbations into ciphertexts stored on the cloud to induce decryption errors on the client side. The client, unaware of the malicious intent, reports these errors to the server. By analyzing the pattern of errors and the timing information associated with homomorphic operations, the attacker can gradually extract the underlying error values for each ciphertext. Specifically, these errors are used to reconstruct a system of linear equations that, when solved, compromise the security of the underlying Learning with Errors (LWE) problem and recover the client's secret key. By strategically inducing errors and using a binary-search approach to recover precise error values, the researchers successfully developed a technique to extract secret keys with a remarkably low number of client queries to avoid detection. By leveraging timing information during homomorphic gate computations, the authors significantly reduce the number of queries needed to extract errors, achieving efficient key recovery. The attack successfully

recovered secret keys for two widely used FHE libraries, FHEW and TFHE, requiring 8 and 23 queries per ciphertext error extraction, respectively. The full-key recovery attack was demonstrated on practical scenarios with TFHE (key size: 630 bits) and FHEW (key size: 500 bits), involving 19,838 and 7,565 client queries, respectively.

11.4 Mitigation Strategies for HE Side-channel Attacks

As mentioned earlier, HE is not immune to side-channel attacks, which exploit implementation-specific vulnerabilities such as timing information, power consumption, or electromagnetic emissions to infer sensitive data. Addressing these threats is critical to ensuring the practicality and trustworthiness of HE systems. This section explores various mitigation strategies designed to harden HE implementations against side-channel attacks and to reduce the attack surface, bolstering the overall security of HE-based applications.

11.4.1 Constant-Time Implementations

Constant-time implementations are a cornerstone of secure cryptographic design, addressing timing side-channel vulnerabilities by ensuring that operations execute in a consistent manner, independent of the input data or secret parameters [130]. Cryptographic algorithms and their implementations are crafted to eliminate timing variations that could inadvertently leak sensitive information, such as secret keys or computational states. This involves uniform execution patterns, where each branch of the code consumes the same computational resources and takes an equal amount of time to complete, regardless of the processed data. By adhering to constant-time principles, developers can significantly reduce the risk of timing attacks, where adversaries analyze variations in execution time to extract critical cryptographic secrets. However, achieving true constant-time behavior can be challenging in complex cryptographic systems, as even minor variations in hardware architectures or compiler optimizations may introduce unintended inconsistencies. Constant-time implementations can vary across platforms. For instance, the "constant-time" fix in OpenSSL designed to mitigate the Lucky Thirteen attack still shows data-dependent execution times on ARM architectures [130] [131]. Additionally, constant-time designs often come with a performance overhead, as they may require additional computations or stricter coding

practices to ensure uniformity. This can impact the efficiency of Homomorphic Encryption systems, where computational performance is already a critical concern.

11.4.2 Masking and Blinding Techniques

Masking and blinding are techniques that randomize intermediate values to mitigate side-channel attacks. When applied to symmetric block ciphers, this is referred to as masking, where data is split into randomized shares. In contrast, when used in public key cryptosystem implementations, it is called blinding, involving the addition of random noise to obfuscate correlations [132]. These techniques introduce random noise or perturbations during cryptographic operations to obscure the computational process and prevent attackers from correlating power traces, timing variations, or other measurable characteristics with secret data. By randomizing internal computations and intermediate values, they aim to reduce the leakage of sensitive information during cryptographic transformations. However, such defenses have significant limitations, particularly their susceptibility to single-trace side-channel attacks [120] [127]. In such scenarios, attackers can exploit advanced statistical or machine learning techniques to bypass the randomness introduced by masking and extract secret information from a single observation. As a result, while masking/blinding can provide a basic level of protection, it is not a recommended standalone defense and should be complemented by more robust strategies to ensure comprehensive security against side-channel threats.

11.4.3 Shuffling and Randomization Techniques

Shuffling and randomization [133] techniques can play a crucial role in mitigating side-channel attacks in the context of Homomorphic Encryption (HE). These methods aim to obfuscate execution patterns and data processing sequences, complicating an attacker's ability to correlate observable side-channel data—such as power consumption, electromagnetic emanations, or timing variations—with sensitive cryptographic operations [134].

- **Shuffling:** Shuffling involves executing operations or processing data in a randomized order [133]. For example, in HE systems that handle multiple ciphertexts or compute on batched data, shuffling the order of operations can disrupt predictable patterns that attackers rely on to analyze side-channel data. This technique is particularly useful in

thwarting statistical side-channel attacks, where repeated patterns are exploited over multiple traces to infer secret information.

- **Random Dummy Operations:** Inserting random, non-functional operations (dummy computations) during cryptographic processing adds noise to power or timing traces, making it harder for attackers to distinguish real computations [134]. In HE, dummy operations must be designed to avoid disrupting the correctness of computations, as the deterministic nature of HE schemes leaves little room for errors introduced by excessive obfuscation. Practical implementations need to balance security gains with the additional computational burden.

- **Randomized Noise Addition:** Adding random noise to intermediate values during HE operations can obscure side-channel data, reducing the risk of pattern detection. However, in HE systems, this noise must be carefully managed to avoid interfering with the decryption process or exceeding the noise budget inherent to HE ciphertexts. Unlike masking, which binds randomness to cryptographic parameters, randomized noise in HE should be lightweight and consistent with the system's correctness constraints.

- **Obfuscated Memory Access Patterns:** Randomizing memory access patterns prevents attackers from exploiting cache-timing or memory-based side-channel vulnerabilities [135]. In HE implementations, this could involve accessing memory blocks in a randomized order or using constant-time memory access strategies to eliminate data-dependent variations. For example, during ciphertext storage or retrieval, randomized access can ensure that memory usage does not reveal sensitive information, albeit at the cost of increased memory latency.

While these techniques significantly enhance side-channel resilience, their implementation in HE systems introduces trade-offs between security and performance. HE operations are computationally intensive, and adding randomization mechanisms can further increase processing time and resource usage. As such, their application should be guided by careful profiling and evaluation of security versus efficiency.

Chapter 12

Additional Research

12.1 Introduction

This chapter provides some ideas on future research ideas.

12.2 Non-lattice based homomomorphic encryption

While most FHE methods use lattice methods, there is still a worry that the LWE problem may be cracked. Thus more traditional methods of using RSA and Paillier provide alternatives for partial HE. With Paillier [38] we can add, subtract and scale multiply, while the RSA method [40] allows us to multiply. To cope with large modulus values, Reddy et al [136] outline a Particle Swarm Optimization (PSO) method (Figure 12.1).

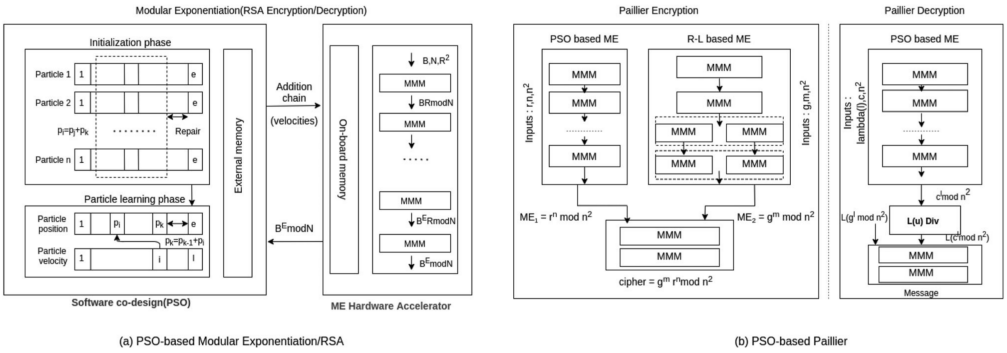

Figure 12.1: PSO Modulo Processing [136]

Bibliography

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure compuation*, pp. 160–179, 1978.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, 2 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[3] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[4] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, ser. EUROCRYPT'99. Springer-Verlag, pp. 223–238.

[5] S. Goldwasser and S. Micali, "Probabilistic encryption," vol. 28, no. 2, pp. 270–299.

[6] T. Sander, A. Young, and M. Yung, "Non-interactive cryptocomputing for nc/sup 1/," in *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pp. 554–566.

[7] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Theory of Cryptography*, J. Kilian, Ed. Springer Berlin Heidelberg, pp. 325–341.

[8] C. Gentry, "A fully homomorphic encryption scheme," aAI3382729.

[9] J. Hoffstein, J. Pipher, and J. H. Silverman, "Ntru: A ring-based public key cryptosystem," in *Algorithmic Number Theory*, J. P. Buhler, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 267–288.

[10] D. Micciancio, "Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions," in *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, 2002, pp. 356–365.

[11] C. Peikert and A. Rosen, "Lattices that admit logarithmic worst-case to average-case connection factors," in *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 478–487.

[12] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology – EUROCRYPT 2011*, K. G. Paterson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 129–148.

[13] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Springer Berlin Heidelberg, pp. 24–43.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 309–325.

[15] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, 2011, pp. 97–106.

[16] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, 9 2009.

[17] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23.

[18] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144. [Online]. Available: https://eprint.iacr.org/2012/144

[19] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 868–886.

[20] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Advances in Cryptology – EUROCRYPT 2012*, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 465–482.

[21] J. H. Cheon, J. Jeong, and C. Lee, "An algorithm for ntru problems and cryptanalysis of the ggh multilinear map without a low-level encoding of zero," *LMS Journal of Computation and Mathematics*, vol. 19, no. A, p. 255–266, 2016.

[22] M. Albrecht, S. Bai, and L. Ducas, "A subfield lattice attack on overstretched ntru assumptions," in *Proceedings, Part I, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9814*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 153–178.

[23] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds. Springer Berlin Heidelberg, pp. 75–92.

[24] Z. Brakerski and V. Vaikuntanathan, "Lattice-based fhe as secure as pke," in *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science*, ser. ITCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–12.

[25] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 297–314.

[26] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640.

[27] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in

Advances in Cryptology – ASIACRYPT 2016, J. H. Cheon and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 3–33.

[28] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Springer International Publishing, pp. 409–437.

[29] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," in *Advances in Cryptology – EUROCRYPT 2021*, A. Canteaut and F.-X. Standaert, Eds. Cham: Springer International Publishing, 2021, pp. 648–677.

[30] J. H. Cheon, S. Hong, and D. Kim, "Remark on the security of CKKS scheme in practice," Cryptology ePrint Archive, Paper 2020/1581, 2020. [Online]. Available: https://eprint.iacr.org/2020/1581

[31] R. Rothblum, "Homomorphic encryption: From private-key to public-key," in *Theory of Cryptography*, Y. Ishai, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 219–234.

[32] C. Gentry, S. Halevi, and V. Vaikuntanathan, "i-hop homomorphic encryption and rerandomizable yao circuits," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 155–172.

[33] M. Naor and M. Yung, "Public-key cryptosystems provably secure against chosen ciphertext attacks," in *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, ser. STOC '90. Association for Computing Machinery, pp. 427–437.

[34] D. Dolev, C. Dwork, and M. Naor, "Non-malleable cryptography," in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '91. Association for Computing Machinery, pp. 542–552.

[35] J. Loftus, A. May, N. P. Smart, and F. Vercauteren, "On cca-secure somewhat homomorphic encryption," in *Selected Areas in Cryptography*, A. Miri and S. Vaudenay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 55–72.

[36] F. Armknecht, S. Katzenbeisser, and A. Peter, "Group homomorphic encryption: characterizations, impossibility results, and applications," *Designs, Codes and Cryptography*, vol. 67, no. 2, pp. 209–232, 5 2013.

[37] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[38] W. J. Buchanan, "Paillier - partial homomorphic encryption," https://asecuritysite.com/paillier/, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/paillier/

[39] ——, "Partial homomorphic encryption with elgamal (multiply)," https://asecuritysite.com/homomorphic/go_el_homo, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/homomorphic/go_el_homo

[40] ——, "Multiplication with homomorphic encryption using rsa," https://asecuritysite.com/homomorphic/hom_rsa, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/homomorphic/hom_rsa

[41] ——, "Polynomials operations (mod p)," https://asecuritysite.com/encryption/poly4, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/encryption/poly4

[42] ——, "Multiplying by the inverse of a polynomial," https://asecuritysite.com/principles/poly_2, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/principles/poly_2

[43] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[44] W. J. Buchanan, "Learning with errors," https://www.youtube.com/embed/hN5TQiz2gWs, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://www.youtube.com/embed/hN5TQiz2gWs

[45] D. Stebila, "Introduction to post-quantum cryptography and learning with errors," https://summerschool-croatia.cs.ru.nl/2018/slides/Introduction%20to%20post-quantum%20cryptography%

20and%20learning%20with%20errors.pdf, Asecuritysite.com, 2018, accessed: September 03, 2024. [Online]. Available: https://summerschool-croatia.cs.ru.nl/2018/slides/Introduction%20to%20post-quantum%20cryptography%20and%20learning%20with%20errors.pdf

[46] W. J. Buchanan, "Finite fields," https://asecuritysite.com/gf/finite, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/gf/finite

[47] R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[48] W. J. Buchanan, "Rsa - partially homomorphic cryptosystem: Division," https://asecuritysite.com/homomorphic/h_rsa2, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/homomorphic/h_rsa2

[49] C. Gentry, "A fully homomorphic encryption scheme," 2009, crypto.stanford.edu/craig.

[50] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer, 2010, pp. 24–43.

[51] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on computing*, vol. 43, no. 2, pp. 831–871, 2014.

[52] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.

[53] W. J. Buchanan, "Homomorphic encryption (seal)," https://asecuritysite.com/seal, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/seal

[54] ——, "Encrypting integers with bgv and bfv using seal and c#," https://asecuritysite.com/seal/seal_01, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/seal/seal_01

[55] ——, "Homomorphic encryption with bfv using node.js," https://asecuritysite.com/seal/js_homomorphic, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/seal/js_homomorphic

[56] A. Wood, K. Najarian, and D. Kahrobaei, "Homomorphic encryption for machine learning in medicine and bioinformatics," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–35, 2020.

[57] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.

[58] A. Al Badawi and Y. Polyakov, "Demystifying bootstrapping in fully homomorphic encryption," *Cryptology ePrint Archive*, 2023.

[59] W. J. Buchanan, "Chebyshev approximations using openfhe and c++ (logarithm methods)," https://asecuritysite.com/openfhe/openfhe_18cpp, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_18cpp

[60] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[61] W. J. Buchanan, "Simple homomorphic cipher for 2-bit adder," https://asecuritysite.com/homomorphic/hom_adder, Asecuritysite.com, 2024, accessed: September 03, 2024. [Online]. Available: https://asecuritysite.com/homomorphic/hom_adder

[62] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer, 2016, pp. 3–33.

[63] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[64] W. J. Buchanan, "Boolean circuits with openfhe using the dm (fhew) and bootstrapping method (ginx) and openfhe," https://asecuritysite.com/openfhe/openfhe_09cpp, Asecuritysite.com, 2024, accessed: September 05, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_09cpp

[65] N. Gama, M. Izabachène, P. Q. Nguyen, and X. Xie, "Structural lattice reduction: generalized worst-case to average-case reductions and homomorphic cryptosystems," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 2016, pp. 528–558.

[66] J. Alperin-Sheriff and C. Peikert, "Faster bootstrapping with polynomial error," in *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer, 2014, pp. 297–314.

[67] Y. Lee, D. Micciancio, A. Kim, R. Choi, M. Deryabin, J. Eom, and D. Yoo, "Efficient fhew bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 227–256.

[68] K. Munjal and R. Bhatia, "A systematic review of homomorphic encryption and its contributions in healthcare industry," *Complex & Intelligent Systems*, vol. 9, no. 4, pp. 3759–3786, 2023.

[69] O. Kocabas, T. Soyata, J.-P. Couderc, M. Aktas, J. Xia, and M. Huang, "Assessment of cloud-based health monitoring using homomorphic encryption," in *2013 ieee 31st international conference on computer design (iccd)*. IEEE, 2013, pp. 443–446.

[70] R. Geva, A. Gusev, Y. Polyakov, L. Liram, O. Rosolio, A. Alexandru, N. Genise, M. Blatt, Z. Duchin, B. Waissengrin *et al.*, "Collaborative privacy-preserving analysis of oncological data using multiparty homomorphic encryption," *Proceedings of the National Academy of Sciences*, vol. 120, no. 33, p. e2304415120, 2023.

[71] S. Halder and T. Newe, "Enabling secure time-series data sharing via homomorphic encryption in cloud-assisted iiot," *Future Generation Computer Systems*, vol. 133, pp. 351–363, 2022.

[72] S. Qi, J. Wang, M. Miao, M. Zhang, and X. Chen, "Tinyenc: Enabling compressed and encrypted big data stores with rich query support," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 176–192, 2021.

[73] M. S. Rahman, A. Alabdulatif, and I. Khalil, "Privacy aware internet of medical things data certification framework on healthcare blockchain of 5g edge," *Computer Communications*, vol. 192, pp. 373–381, 2022.

[74] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*. Springer, 2011, pp. 578–595.

[75] W. J. Buchanan, "Order preserving encryption," https://medium.com/asecuritysite-when-bob-met-alice/order-preserving-encryption-ope-5695504cfda3, Medium.com, 2024, accessed: September 04, 2024. [Online]. Available: https://medium.com/asecuritysite-when-bob-met-alice/order-preserving-encryption-ope-5695504cfda3

[76] Z. Shi, Z. Yang, A. Hassan, F. Li, and X. Ding, "A privacy preserving federated learning scheme using homomorphic encryption and secret sharing," *Telecommunication Systems*, vol. 82, no. 3, pp. 419–433, 2023.

[77] W. J. Buchanan, "Proxy re-encryption (pre) with ckks homomorphic encryption using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_21cpp, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_21cpp

[78] ——, "Paillier adding and multiplying with kryptology," https://asecuritysite.com/paillier/pail01, Asecuritysite.com, 2024, accessed: October 30, 2024. [Online]. Available: https://asecuritysite.com/paillier/pail01

[79] ——, "Homomorphic encryption (openfhe)," https://asecuritysite.com/openfhe, Asecuritysite.com, 2024, accessed: October 30, 2024. [Online]. Available: https://asecuritysite.com/openfhe

[80] I. Amorim and I. Costa, "Leveraging searchable encryption through homomorphic encryption: A comprehensive analysis," *Mathematics*, vol. 11, no. 13, p. 2948, 2023.

[81] T. Altuwaiyan, *Toward Data Privacy Related to the Internet of Things*. University of Massachusetts Boston, 2018.

[82] J. Bell, D. Butler, C. Hicks, and J. Crowcroft, "Tracesecure: Towards privacy preserving contact tracing," *arXiv preprint arXiv:2004.04059*, 2020.

[83] H. Cho, D. Ippolito, and Y. W. Yu, "Contact tracing mobile apps for covid-19: Privacy considerations and related trade-offs," *arXiv preprint arXiv:2003.11511*, 2020.

[84] T. Kim, Y. Oh, and H. Kim, "Efficient privacy-preserving fingerprint-based authentication system using fully homomorphic encryption," *Security and Communication Networks*, vol. 2020, no. 1, p. 4195852, 2020.

[85] H. Asi, F. Boemer, N. Genise, M. H. Mughees, T. Ogilvie, R. Rishi, G. N. Rothblum, K. Talwar, K. Tarbe, R. Zhu *et al.*, "Scalable private search with wally," *arXiv preprint arXiv:2406.06761*, 2024.

[86] Apple, "Combining machine learning and homomorphic encryption in the apple ecosystem," https://machinelearning.apple.com/research/homomorphic-encryption, Apple, 2024, accessed: December 04, 2024. [Online]. Available: https://machinelearning.apple.com/research/homomorphic-encryption

[87] A. C.-C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*. Chicago, IL, USA: IEEE, 1982, pp. 160–164.

[88] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Advances in Cryptology – CRYPTO 2012*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, Berlin, Heidelberg, 2012, pp. 643–662.

[89] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[90] Y. Desmedt, "Threshold cryptosystems," in *Advances in Cryptology – AUSCRYPT '92*, ser. Lecture Notes in Computer Science, J. Seberry

and Y. Zheng, Eds., vol. 718. Springer, Berlin, Heidelberg, 1993, pp. 450–457.

[91] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *28th Annual Symposium on Foundations of Computer Science (SFCS 1987)*. Los Angeles, CA, USA: IEEE, 1987, pp. 427–438.

[92] W. J. Buchanan, "Bfv with threshold encryption using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_16cpp, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_16cpp

[93] M. Iezzi, "Practical privacy-preserving data science with homomorphic encryption: an overview," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 3979–3988.

[94] W. J. Buchanan, "Logistic (sigmoid) function evaluation using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_20cpp, Asecuritysite.com, 2024, accessed: September 06, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_20cpp

[95] R. Bender and U. Grouven, "Ordinal logistic regression in medical research," *Journal of the Royal College of physicians of London*, vol. 31, no. 5, p. 546, 1997.

[96] W. J. Buchanan, "Ckks inner product using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_13cpp, Asecuritysite.com, 2024, accessed: September 05, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_13cpp

[97] ——, "Matrix multiplication with homomomorphic encryption for bfv using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_26cpp, Asecuritysite.com, 2024, accessed: September 06, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_26cpp

[98] M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser, "Secure large-scale genome-wide association studies using homomorphic encryption," *Proceedings of the National Academy of Sciences*, vol. 117, no. 21, pp. 11 608–11 613, 2020.

[99] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[100] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[101] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, p. 1484–1509, oct 1997.

[102] M. Mosca, "Cybersecurity in an era with quantum computers: Will we be ready?" *IEEE Security & Privacy*, vol. 16, no. 5, pp. 38–41, 2018.

[103] S. Beauregard, "Circuit for shor's algorithm using 2n+3 qubits," *Quantum Info. Comput.*, vol. 3, no. 2, p. 175–185, mar 2003.

[104] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," 2016-04-28 2016.

[105] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 922–934.

[106] W. Jung, E. Lee, S. Kim, N. Kim, K. Lee, C. Min, J. H. Cheon, and J. H. Ahn, "Accelerating fully homomorphic encryption through microarchitecture-aware analysis and optimization," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 237–239.

[107] S. Fu, N. Zhang, and F. Franchetti, "Accelerating high-precision number theoretic transforms using intel avx-512."

[108] N. Zhang and F. Franchetti, "Generating number theoretic transforms for multi-word integer data types," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2023.

[109] A. Ş. Özcan, C. Ayduman, E. R. Türkoğlu, and E. Savaş, "Homomorphic encryption on gpu," *IEEE Access*, vol. 11, pp. 84 168–84 186, 2023.

[110] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[111] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[112] D. Soni, N. Neda, N. Zhang, B. Reynwar, H. Gamil, B. Heyman, M. Nabeel, A. Al Badawi, Y. Polyakov, K. Canida *et al.*, "Rpu: The ring processing unit," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 272–282.

[113] C. Castelluccia, E. Mykletun, and G. Tsudik, "Efficient aggregation of encrypted data in wireless sensor networks," in *The second annual international conference on mobile and ubiquitous systems: networking and services*. IEEE, 2005, pp. 109–117.

[114] K. Kapusta, G. Memmi, and H. Noura, "Additively homomorphic encryption and fragmentation scheme for data aggregation inside unattended wireless sensor networks," *Annals of Telecommunications*, vol. 74, no. 3, pp. 157–165, 2019.

[115] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[116] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina, "Sok: Deep learning-based physical side-channel analysis," *ACM Computing Surveys (ACM Comput. Surv.)*, vol. 55, no. 11, pp. Article 227, 35 pages, November 2023. [Online]. Available: https://doi.org/10.1145/3569577

[117] M. Devi and A. Majumder, "Side-channel attack in internet of things: A survey," in *Applications of Internet of Things*, ser. Lecture Notes in Networks and Systems, J. Mandal, S. Mukhopadhyay, and A. Roy, Eds. Springer, Singapore, 2021, vol. 137, pp. 257–270. [Online]. Available: https://doi.org/10.1007/978-981-15-6198-6_20

[118] D. Strobel, I. C. Paar, and M. Kasper, "Side channel analysis attacks on stream ciphers," *Masterarbeit Ruhr-Universität Bochum, Lehrstuhl Embedded Security*, 2009.

[119] Audriusa (Wikimedia Commons), "Oscilloscope reading showing power consumption variations," 2024, licensed under the GNU

Free Documentation License (GFDL). [Online]. Available: https://en.wikipedia.org/wiki/File:Image.png

[120] F. Aydin, E. Karabulut, S. Potluri, E. Alkim, and A. Aysu, "Reveal: Single-trace side-channel leakage of the seal homomorphic encryption library," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Antwerp, Belgium, 2022, pp. 1527–1532.

[121] R. Onishi, T. Suzuki, S. Sakai, and H. Yamana, "Security and performance-aware cloud computing with homomorphic encryption and trusted execution environment," in *Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 36–42. [Online]. Available: https://doi.org/10.1145/3689945.3694805

[122] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, Firstquarter 2018.

[123] F.-X. Standaert, *Introduction to Side-Channel Attacks*. Boston, MA: Springer US, 2010, pp. 27–42. [Online]. Available: https://doi.org/10.1007/978-0-387-71829-3_2

[124] T. Messerges, "Using second-order power analysis to attack dpa resistant software," in *Cryptographic Hardware and Embedded Systems – CHES 2000*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2000, vol. 1965, pp. 238–251. [Online]. Available: https://doi.org/10.1007/3-540-44499-8_19

[125] F. Aydin and A. Aysu, "Leaking secrets in homomorphic encryption with side-channel attacks," *Journal of Cryptographic Engineering (J Cryptogr Eng)*, vol. 14, pp. 241–251, 2024. [Online]. Available: https://doi.org/10.1007/s13389-023-00340-2

[126] P. Ravi, S. Bhasin, S. S. Roy, and A. Chattopadhyay, "On exploiting message leakage in (few) nist pqc candidates for practical message recovery attacks," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 684–699, 2022.

[127] W. Cheng, J.-L. Danger, S. Guilley, F. Huang, A. B. Korchi *et al.*, "Cache-timing attack on the seal homomorphic encryption library," in

11th International Workshop on Security Proofs for Embedded Systems (PROOFS 2022), Leuven, Belgium, Sep 2022. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03780506

[128] R. Primas, P. Pessl, and S. Mangard, "Single-trace side-channel attacks on masked lattice-based encryption," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, ser. Lecture Notes in Computer Science, W. Fischer and N. Homma, Eds., vol. 10529. Springer, Cham, 2017, pp. 537–557. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_25

[129] B. Chaturvedi, A. Chakraborty, A. Chatterjee, and D. Mukhopadhyay, "A practical full key recovery attack on tfhe and fhew by inducing decryption errors," *Cryptology ePrint Archive*, 2022. [Online]. Available: https://eprint.iacr.org/2022/1563

[130] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security (J Hardw Syst Secur)*, vol. 2, pp. 33–50, 2018. [Online]. Available: https://doi.org/10.1007/s41635-017-0025-y

[131] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on sel4," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 570–581. [Online]. Available: https://doi.org/10.1145/2660267.2660294

[132] X. Hou and J. Breier, "Side-channel analysis attacks and countermeasures," in *Cryptography and Embedded Systems Security*. Springer, Cham, 2024. [Online]. Available: https://doi.org/10.1007/978-3-031-62205-2_4

[133] S. Patranabis, D. B. Roy, P. K. Vadnala, D. Mukhopadhyay, and S. Ghosh, "Shuffling across rounds: A lightweight strategy to counter side-channel attacks," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 440–443.

[134] C. Liptak, S. Mal-Sarkar, and S. A. P. Kumar, "Power analysis side channel attacks and countermeasures for the internet of things," in *2022 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. IEEE, 2022, pp. 1–7. [Online]. Available: https://doi.org/10.1109/PAINE56030.2022.10014854

[135] Z. H. Jiang, Y. Fei, A. A. Ding, and T. Wahl, "Mempoline: Mitigating memory-based side-channel attacks through memory access obfuscation," *Cryptology ePrint Archive*, 2020. [Online]. Available: https://eprint.iacr.org/2020/760.pdf

[136] S. S. Reddy, S. Sinha, and W. Zhang, "Design and analysis of rsa and paillier homomorphic cryptosystems using pso-based evolutionary computation," *IEEE Transactions on Computers*, vol. 72, no. 7, pp. 1886–1900, 2023.